Management of Software Inspection Process: A Learning Tool

A Thesis submitted
In Partial fulfillment of the Requirements
For the degree of

MASTER OF TECHNOLOGY

By
Ambrish Kumar Srivastava
(9911402) M.Tech. [IME]

DEPARTMENT OF INDUSTRIAL AND MANAGEMENT ENGINEERI
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

March 2001.

CERTIFICATE

This is to certify that the present dissertation entitled "**Management of Software Inspection: A Learning Tool**" has been carried out by Mr. Ambrish Kumar Srivastava (Roll No. 9911402) under my supervision and that this work has not been submitted elsewhere for a degree.

27 March, 2001

Dr. A. K. Mittal
Professor,
Deptt. Of Industrial and Management Engg.
Indian Institute of Technology
Kanpur-208016
India.

ACKNOWLEDGEMENTS

With great pleasure, I express my indebt ness to Professor A. K. Mittal for his invaluable help and guidance throughout the course of this work. I am heartily thankful to him for the confidence he showed in me. It was also his special guidance and nature that made working on this thesis informative, enjoyable, and a special learning experience. I would also like to thank Mr. Vivek Bhagwat Ex Alumni of our department, and my colleague Ritesh Chausria, for their invaluable help and encouragement.

I would further like to thank all faculty member of IME department who are always there with helpful suggestions and provide ample encouragement.

Last but not least a special gratitude to almighty who always be a source of inspiration to me.

- Ambrish Kumar Srivastava

ABSTRACT

Inspections of software development work products have become an integral part of software quality improvement. However, experience tells, that inspection effectiveness (i.e. it's capability to find a defects) and its cost effectiveness vary significantly across organizations or even more striking, from one project to another in the same organization. Some managers possess an intuitive latent understanding of the requirements for a better management of inspection, and make decision using the skills and their experience. However, when wrong decision are made and implemented into a inspection process, disastrous result may occur, increasing the cost of finding the defect at the letter stages of testing or even delivering the product with lower quality level.

In this thesis attempt is made to develop a 'Management Game' a learning tool for the software inspection process, which captures various factor, that have an impact on the effectiveness and cost of the inspection process, and that can explain most of their variation.

The relationships used to develop the model represent, casual influence rather than statistical correlation. These relationships take into consideration variables like experience level of inspectors, checking rate, complexity of the work product, team size etc.

This tool can be used to train the software project manager to develop skills for allocation of inspection effort, taking into consideration the various factors, which affect the quality and the cost of inspection. Using several projects of different complexity level, and iteratively experimenting with different inspection allocation, the trainee can learn the tactics and skills needed for software inspection effort allocations.

This tool can be used to simulate the decision taken by the users (players), about the team size, the checking rate for the work product of different complexity level, etc. and the outcome of various decisions can be observed and analyzed.

Contents

List of Figures

List of Tables

CHAPTER ONE.

Introduction

1.1 Overview

From the historical perspective, the 1960s and the year prior to that decade could be viewed as the functional era of software engineering, the 1970s as the scheduling era, and the 1980s as the cost era [Basili 1991]. In the 1960s the software engineering community learned how to exploit information technology to meet institutional needs, and began to link the software with the daily operations of institutions in the society. In the 1970s, when software development was characterized by massive schedule delays and cost over-runs, the focus was on planning and control of software projects. In the effort to bring discipline to the development of the software system attempts have been made since the early 1970s to apply the rigors of science and engineering to the software production process. This leads to significant advances in the technology of software production i.e.; Phase based life cycle models were introduced, and analyses like the mythical man month emerged. In the 1980s, the hardware costs continued to decline. Information technology permeated every facet of organizations. As competition in the computer industry became keen and low-cost applications became widely implemented, the importance of productivity in software development increased significantly. Various cost models in software engineering were developed and used. In the late 1980s, the importance of quality was also recognized. The managerial aspects of software development, on the other hand, have attracted much less attention from the research community [Zumud, 1980]. Copper [Copper, 1978] provides an insightful explanation for the reason why: Perhaps this is so because computer scientist believes that management per se is not their business, and the management professional assumes that, it is the computer scientist responsibility.

The 1990s and beyond is certainly the quality era. As state-of-the-art technology is now able to provide abundant functionality, customer demands high quality. Demand for quality is further intensified by the ever-increasing dependence of our society on software. Billing errors, large-scale disruption of telephone services, and even a missile

failure during the gulf war at 1992 can all be 'traced' to the issue of software quality [Littlewood 1992]. In this era, quality has been brought to the center of software development process. Many companies and researcher in different domains started focusing on the issue of quality. As the role of software is expanding in many aspects of modern life, quality and customer satisfaction becomes the main goal for the software developers and an important market value for organization. A conservative estimate indicates, a hundred fold increase in demand for software in the last two decade [Musa, 1985]. The growth of the software has not been painless. The records show that the developers of the software have been marked by cost over-run, late deliveries, poor reliability and user dissatisfaction [Meyer, 1995].

As Grady [Grady, 1993] shows "quality is vital in competitiveness in today's world. How can we know if have achieved quality? we want to build quality form the beginning- how can we ensure that is done?". Software developers must take into account these concern and as Haag [Haag, 1991] points out, "quality is even more important than productivity of software, establishing product quality and development efficiency as equally important".

Unfortunately, neither how to ensure quality nor how much it will cost is known. Sometime quality is not even well defined. Some people defined quality as zero defects in the delivered software. Another approach is, that quality means, "to satisfy the customer". The quality is defined by set of attributes required by the customers and specified in the nonfunctional requirements. These are the so-called 'ilities' such as reliability, usability, maintainability, portability, security, and safety.

The basic idea of achieving the software quality attributes is that quality of the product depends on process followed to produce it. The claim is that by improving the development process the resulting software will have a high quality.

As demands for software have outstripped our production capability, an increasing amount of attention has been placed on improving the software development process. The

software development process, as defined by Humphery [Humphrey, 1995], is the set of tools, methods, and practices we used to produce a software product. The objective of software process improvement, then, is to produce products according to the plan, while simultaneously improving organization capabilities to reach there ultimate goals of producing products of higher quality, at lower cost, in less time. A process improvement will not necessarily achieve all three goals, but when attention is paid to the quality control, a reduction in cycle time and cost are likely to follow. The realization that process improvement can benefit software development organizations has led to the process improvement models, such as Software Engineering Institute's Capability Maturity Model (SEI CMM) [Paulk, 1993] and ISO 9000, 9002.

The SEI CMM grew out of work; the SEI was performing for the U.S. Air Force. The objective was to provide the military with a method for selecting capable contractors. The resulting method proves useful for assessing other software development organizations, and as a result of these assessment, they identified technical and managerial topics considered most critical to process improvement [Humphrey, 1995].

The five level of SEI CMM 'distinct and tangible were clearly scribed as:

- **Initial:** Proceed from chaotic to predictable quality cost and schedule.
- **Repeatable:** Progress from intuitive to known standards of procedure and methods.
- **Defined:** Advances from qualitative to measure method of quality management.
- **Managed:** Improved from measured to statistical quality control.
- **Optimizing:** Continue incorporation of effective development procedure

Level	Focus	Key Process Areas
5 Optimizing	*Continuous process improvement*	Acquisition Innovation Management Continuous Process Improvement
4 Quantitative	*Quantitative management*	Quantitative Acquisition Management Quantitative Process Management
3 Defined	*Process standardization*	Training Program Acquisition Risk Management Contract Performance Management Project Performance Management Process Definition and Maintenance
2 Repeatable	*Basic project management*	Transition to Support Evaluation Contract Tracking and Oversight Project Management Requirements Development and Management Solicitation Software Acquisition Planning
1 Initial	*Competent people and heroics*	

Figure. 1.1 SEI CMM Levels [Bate, 1995]

Improving the software development process improves the quality of the software products and the over all performance, of the software development organization [Humphrey, 1995].

However as shown in figure 1.2, process is only one of the several controllable factors in improving software quality and organization performance, other include, the skills and experience of the developers, the technology used, product complexity, and environmental characteristics such as schedule pressure and communications [Kellener, 1992]

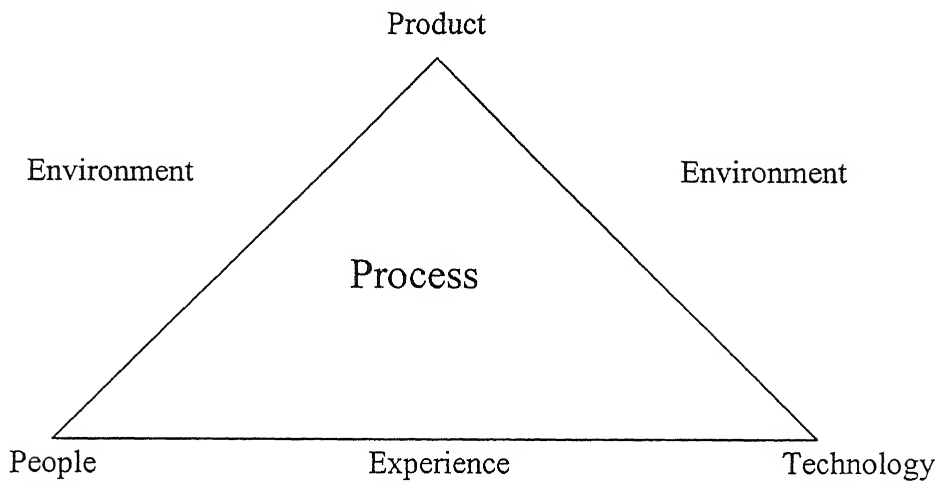


Figure 1.2 Software Quality Determinants [Anita, 1994]

Software quality assurance constitute a set of activities undertaken during the development of software system so as to reduce the risk of unacceptable system performance and to ensure that the produced software does conform to establish technical requirements [Pressman, 1982]. Software quality assurance can be approached through two complementary strategies. First, through ensuring, that the quality is initially built into the product. This involves ensuring the early generation of a coherent, complete, unambiguous, and non-conflicting set of user requirements. Then, as the product moves into the design and coding stages, second sets of QA tools are deployed to continuously inspect and test the system. QA activities now become an integral part of the software development process.

1.2 Problem Statement

One of the most challenging and significant avenues of research in software engineering discipline is the investigation of how to ensures software quality, reduce the development cost and keep software projects within the schedule. Software inspection is practical approach to tackle all three issues. However, there still exist challenging questions, which software organization needs to be able to answer:

1. What is the most cost effective variation?

2. How to assign resources to an inspection in an optimal way?
3. How does the number and experience of inspectors influence software inspection?
4. The percent of error found during inspection?
5. How much to inspect?

There will always be a point at which overspending on QA (quality assurance) activity will be "wasteful" from a project life cycle point of view in the sense that it would be more cost effective to defer the detection of the remaining errors to the testing phase.
"In any sizeable program, it is impossible to remove all error (during development)... some error-manifested themselves, and can be exhibited only after system integration" [Shooman, 1983].

The most successful inspection approach is the one, which helps to find most of the defects in the inspected artifact, has an optimal cost/benefit ratio and can be performed within specified time frame.

1.3 Proposed Learning Tool for Software Inspection Process:

In this theses attempt is made to correlate above identified questions with variables connected to the software inspection process. This require first developing a model which includes information concerning process steps of software inspection, behavior of inspectors and other product attributes, and also the selection of inspection metrics which reflects the inspection process from it's depth.

The overall approach is to develop a management game, for the software inspection process, which may be executed to simulate inspection process of project under development, under a set of specified assumptions.

Such on learning tool is expected to make following contributions:

a) A simulation game is an important training tool and will help the software developers and student in learning different aspects of software inspection.

b) An executable model of a software inspection process will give researchers and software project managers at software organization the ability to answer questions and make informed decisions. An executable model will allow controlled experiment to examine different strategies to examine cost/benefit ratio of inspection process, thus providing a more scientific approach to the decision making process. Importantly, an executable model may be used as management simulator to provides hand on training to illustrate the advantage and disadvantage for using the inspection process.

c) An executable model requires that metrics are to be collected and entered into the model, in order to simulate a project. The metrics required by the model, define a metric set for collection by any software development organization, wishing to begin a process improvement program. Once collected, the metric values may be entered into the model and "played back" for analysis. The reward for collecting this set of metrics can result in more effective software inspection process.

d) To extend the concept of system dynamics model, that helps in developing model for the software development process for the future research.

e) A model specification, which provides a foundation for discussion and debate on the factors that, affects the software inspection process.

1.4 Organization of the Theses:

Chapter two entitled **Introduction to Software Development and Inspection Process** provides a classification of software development process and discusses in detail the various dimensions/sub dimensions associated with the inspection process. In this chapter various variants to conventional approach of software inspection process are also discussed.

Chapter three entitled **Quality Management Models** discusses current approaches for evaluating the quality level and also give brief idea of the work that has been done so far in the related area and is significant to the development of learning tool.

Chapter four entitled **Designing a Learning Tool for Management of Software Inspection Process (A management game)** discusses the model for the software inspection process and also provides the specification of the model with respect to various variable associated with the software inspection process.

Chapter five entitled **Model Description and User- Interface** describe model boundary and also specify the tool in details by specifying input and output interfaces of the tool and steps related with the use of the tool.

Chapter six discusses **Conclusion and Direction for the Future work.**

Appendix A specifies definition of the key terms and defect classification scheme.

Appendix B specifies the source code used to develop the 'tool'.

CHAPTER TWO.

Introduction to Software Development and Inspection Process

In this chapter software development and inspection process is discussed. The first section introduces the software development process. The second section discusses in details the software inspection process and also describes the various dimension and sub dimensions. Various set of metrics used to evaluate the inspection process and their suggested value are also discussed. It also discusses variants to conventional inspection process.

2.1 Development Process

In a broad way, process can be defined as task, that transform some inputs to outputs. The goal is to specify a process that will perform the task in an optimal manner. Since development of large software is a complex task, divide and conquer approach is likely to be more effective in developing the software.

The clear emphasis in the modern approach to software engineering is to focus attention on the overall development process. This is the aim of structured software development, which break downs the development process into a series of distinct phases, each with well defined objective, the output of which can be verified, ensuring a sound foundation for the succeeding phase. This allows overall planning of 'how' the software is going to be developed, as well as considering 'what' is going to be developed as the product.

2.1.1 A Generic Life Cycle Model:

For software development, there are several life cycle models in use. The most common are the waterfall model, iterative enhancement model, prototyping model, and spiral model. (Description and limitation of these models can be found in Pressman [Pressman1982]).

Any modern model should be easy to relate to the following phases:

- Problem description
- Requirement specification
- High level design (structural design)
- Low level design (detailed design)
- Implementation (coding)
- Unit test
- Integration (function) test
- System verification test
- Acceptance test

2.1.2 Software development life cycle phases:

Listed below are the major activities that are performed generically:

2.1.2.1 Problem Description:

This document identifies the problem as visualized by the customer. The description might not be restricted to the software aspects of the problem but might also address a broader system context beyond the software component of that system.

2.1.2.2 Requirement Specification Phase:

IEEE define requirement as " A condition of capability needed by a user to solve a problem or achieves an objective [IEEE, 1987]". The requirement phase translates the ideas in the minds of the clients (the input), into a formal document (the output of the requirement phase). Thus during this phase, a specific methodology is used to gather the requirements of a product. These requirements are analyzed, documented, and used to create a solution that describes new and enhanced functionality in response to each requirement.

2.1.2.3 High Level Design (HLD):

HLD defines the product function that will satisfy the specified requirements. These functions are then partitioned into sub structure with hierarchical relationships. Components names are defined, together with their principal data structure and control path. Prototyping may be employed in this stage to assist in exploring various design approaches and to validate the capability of the product to meet the specified requirements.

The details of high-level design of a software product are documented in a design specification document. This document contains a complete and detailed description of all the externals of the product.

2.1.2.4 Low Level Design (LLD):

During LLD, the HLD substructures of a software product are further refined into procedures and their logic paths are detailed. Data structures are defined to the lowest level of details.

The complete design of a software product is documented in a design structure document. This document details the structural and logic aspects of the products by describing the components and modules, their function, data usage and interfaces. The LLD stage is complete when the design structure document has been validated via inspection against the requirements and design specification documents of the product.

2.1.2.5 Implementation:

The completed design of the software product is transformed into a compilable language. The source programs, the individual test cases and test scenarios are then inspected prior to software testing.

2.1.2.6 Unit Testing:

The inspected source code of the software product is unit tested. Unit test are essentially path tests done on a white box basis. White box test examines, the basic design of the program and require that the tester have detailed knowledge of the program internal structure [Pressman, 1982]. The logic of each module is tested to ensure that a high percentage of statements are executed at least once, and similarly a high percentage of branches are traversed at least once. White box tests focus on a relatively small segment of code and aim to examine a high percentage of the internal paths of the code.

2.1.2.7 Function Verification Testing (FVT):

Functional or black tests are designed to verify the program to its external specifications. The product is integrated and all documented product functions are executed to test for conformance with the design specifications [Pressman, 1982].

2.1.2.8 System Verification Test (SVT):

An integrated hardware and software system that emulates the user environments is built, and all documented product functions are executed from the user perspective. Further, the system is stressed from the performance, reliability, and compatibility point of view points.

2.1.2.9 Acceptance and Installation:

Acceptance and installation is the phase in the software lifecycle, during which a software product is integrated, into its operational environment and tested in this environment, to ensure that it performs, as required. This phase includes two basic tasks: getting the software accepted and installing the software at the customer site. Acceptance consists of formal testing conducted by the customer according to the acceptance test plan prepared earlier and analysis of the test result to determine whether the system satisfy its acceptance criteria. Installation involves placing the accepted software in the actual production environment after the system satisfies acceptance criterion [Jalote, 2000].

2.2 Software Inspection

Software inspections are not a new idea. They have been around almost as long as software has, in the form of software review. Yourdan [Yourdan, 1989], defined software inspection as an approach involving a well defined and disciplined process in which a team of qualified personnel analyzes a software product, using a reading technique for the purpose of detecting defects.

The prime objective of the software inspection process is to identify defects so that they can be corrected with in the current development stage and decreases the number of defects transmitted from one development stage to the next, substantially improving the quality of the product.

2.2.1 Related Literature

After Fagan's [Fagan, 1976] seminal introduction of the generic notion of inspection to the software domain at IBM in the early 1970s[Fagan, 1976], a large number of research paper after proposing, new methodologies and/or incremental improvement have been reported in literature promising to leverage and amplify inspection's benefits within software development projects.

Kim. [Kim, 1995] present a framework for software technical development review including software inspection. Article segmented the framework, according to aims and benefit of reviews, human elements, review process, review output and other matters. Macdonald [Macdonald, 1996] describes the scope of support for the currently available inspection process and review tools. Portal [Portal, 1995] focus their attention on the organizational attributes of the software inspection processes, such as, the team size or the number of session to understand how these attributes influences the cost and benefits of software inspection. Wheeler [Wheeler, 1997] discuss the software inspection process as a particular type of peer review process and elaborate the difference between software inspection's walk through, and other peer review processes. Tjahjono [Tjahjono, 1996] presents a framework for formal technical reviews (FTR) including objective, collaboration, roles synchronicity, technique, and entry/exit criteria as dimensions. Tjahjono's aims at determining the similarities and differences between the review processes of different FTR methods, as well as to identify potential review success factors.

2.2.2 Dimensions of software Inspection Process

Some of the relevant question of concern to practitioners of software inspection process are:

- What are key differences among the currently available inspection approaches?
- On which part of life cycle can these approaches be applied?
- What are their documented effects at those stages?
- What category of effects can these approaches have on project, or an organization?
- What are qualitative and quantitative results to support the claims?
- What types of tools are available to support inspections?

Oliver [Oliver, 1998] has attempted to characterize the nature of software inspection in relation to five primary dimensions --- technical, managerial, economics, organizational, and tools. Yet, although necessary, these five primary dimensions are not unique, but have significant relevance to the major areas of software development. Hence, particular

sub dimensions, from the literature that were seen as fundamental to the nature and application of software inspection, have been elicited. This section discusses, each dimension and its associated primary goals in detail using the relevant articles:

2.2.2.1 The Technical Dimension of Software Inspection:

Inspection must be tailored to fit particular development situation. To do so, it is fundamental to characterize the technical dimension to current inspection method, and their refinements to grasp the similarities and differences among them. The technical dimension of software inspection process includes, the inspected product, the team role participants have in an inspection as well as the team size, and the reading techniques as the sub dimensions.

2.2.2.1.1 The Process Dimension:

To explain various similarities and differences among the methods, a reference model for software inspection process is needed. To define such a reference model, the purpose of various activities within an inspection rather then their organization was adhered to. This allows us to provide examination of different approaches. Six major inspection process stages were identified: Planning, Overview, Defect detection, Defect collection, Defect correction, and Follow-up.

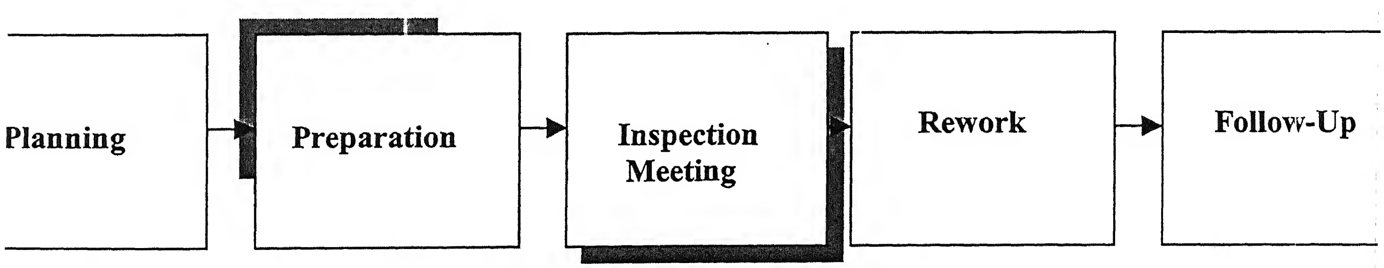


Fig. 2.1, Stages of Inspection Process [Bourgeois, 1996]

2.2.2.1.1.1 Planning

The objective of the planning phases is to organize a particular inspection when material to be inspected pass entry criteria, such as when source code successfully compiles without syntax error. This phase include selection of the inspection participants, their assignment to roles, the scheduling of the inspection meeting and the distribution of the inspection material.

2.2.2.1.1.2 Overview

The overview phase consists of a first meeting in which the author explains the inspected product to other inspection participants. The main goal of the overview phase is to make the inspected product more lucid and, therefore, easier to understand. Such a first meeting can be particularly valuable for the inspection of early artifact, such as, requirements or design document, but also for complex source code.

2.2.2.1.1.3 Defect detection/Defect collection:

The defect detection phase can be considered the core of the inspection. The main goal of the defect detection phase is to scrutinize a software artifact to elicit defect. This phase comprises of two phases, Preparation Phase and Inspection meeting Phase. Generally in the preparation phase, inspectors individually check the design/code document against the functional specification to detect defect. If an inspector detects a defect, it will be classified and documented on a report form. However, the major goal in inspection meeting is to decide, for each defect detected during the individual preparation, whether it is a real defect or a false positive. A false positive is an issue that is logged as defect throughout the individual preparation, but turns out not to be a real defect i.e., affecting the correctness of the document inspected. Additional defects may be detected during meeting and logged in inspection meeting form.

2.2.2.1.1.4 Defect correction:

Throughout the defect correction phase the developer reworks and resolve defects found [Fagan, 1976] or rationalize their existence. For this inspector edits the material and deal with each reported defect.

2.2.2.1.1.5 Follow-up:

The objective of the follow–up phase is to check whether all defects are resolved. For this, one of the inspection participants verifies the defect resolution.

2.2.2.1.2 The Product Dimension:

The product dimension refers to the type of product that is usually inspected. Barry Boehm [Boehm, 1981] stated that one of the most prevalent and costly mistakes made in software projects today is detecting and correcting software defects until late in the projects. This statement supports the use of software inspection for early life cycle documents. However, a look in the literature reveals that in most cases inspection was applied to code documents. Figure 2.2 depicts products inspection taking place between a development cycle phases. The arrow returning to the preceding software development phase reflect the change which are made to the product inspected as a result of correcting defects found in it.

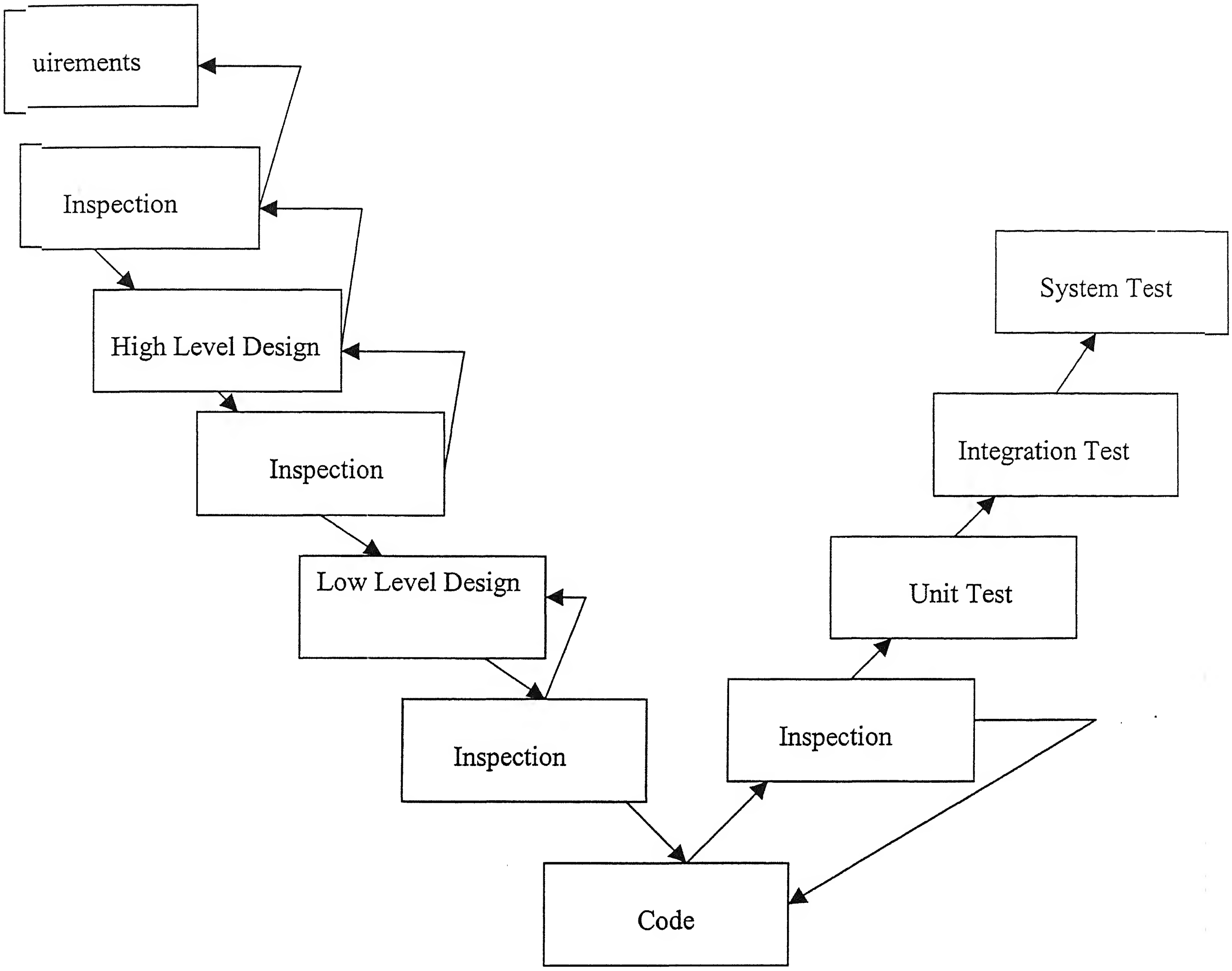


Figure 2.2 Inspection and Development Phases, the 'V' Model
[Glib, 1993]

2.2.2.1.3 The Team Role and Size

Three important questions, practitioners usually face about software inspection are (1) what roles are involved in inspection, (2) how many people to be assigned to each role (3) how to select people for each role. For the first question, a number of specific roles are assigned to inspection participants, so that a participant has clear and specific responsibilities. The role and their responsibilities are described below:

- Organizer: the organizer plans all inspection activities within a project or even across projects
- Moderator: the moderator ensures that inspection procedures are followed and that team member performs their responsibilities for each phase. He also moderate the inspection meeting
- Inspector: Inspectors are responsible for detecting defects in the target software product. Usually all team member can be assumed to be inspector, regardless of their specific role.
- Reader/Presenter: in an inspection meeting, the reader will lead the team through the material in a complete logical fashion.
- Author: the author has developed the inspected product and is responsible for the correction of defects during the rework.
- Recorder: the recorder is responsible for logging all defects in an inspection defect list during the inspection meeting.
- Collector: the collector collect the defect found by inspectors, if there is no inspection meeting.

To answer second question, that is, how to assign resources to these roles in an optimal manner, the reported literature is not uniform. Fagan recommends keeping the inspection team small, that is four people [Fagan, 1976]. Bisant [Bisant, 1989] have found performance advantage in an experiment with two persons: one inspector and the author, which can also be regarded as an inspector. Porter [Portel, 1997b]. Experiment results suggests that reducing the number of inspectors from 4 to 2 may significantly reduce effort, without increasing the inspection interval or reducing effectiveness. There is no particular answer to this question and, an answer heavily depends on the type of product and the environment in which an inspection is performed.

2.2.2.1.4 The Reading Technique Dimension:

A reading technique can be defined as a series of steps or procedures whose purpose is for an inspector to acquire deep understanding of the inspected work product. The comprehension of the inspected work product is a prerequisite for detecting subtle and/or complex defects, which often cause the most problem, if detected in later life cycle phases.

Even though reading technique is one of the key activities for individual defect detection [Basili, 1997], few documented reading technique are currently available to support the activity. Ad-hoc reading and checklist-based reading are probably the most popular reading techniques used today for defect detection in inspections [Gilb, 1993].

Ad-hoc reading, by nature, offers very little reading support at all, since a software product is simply given to inspection, without any direction or guidelines on how to proceed through it, and what to look for. However, Ad-hoc does not mean, that inspection participants do not scrutinizes the inspected product systematically. The word 'Ad-hoc' only refers to the fact, that no support is given to them. In this case, defect detection fully depends on the skill, the knowledge, and the experience of an inspector, which may compensate the lack of reading support.

Checklists offer stronger, support in form of questions that inspector must answer while reading the document. Although reading support, in the form of a list of a question is better than none (such as Ad-hoc), check list-based reading has several weaknesses as: first, the questions are often general and not sufficient tailored to a particular development environment. Thus, the checklist provides little support to help an inspector to understand the inspected artifact. This can be vital to detect, application logic defects. Second, concrete instructions, on how to use check list is often missing. That is, it is often unclear when and based on what task information an inspector is to answer a particular checklist question. Finally, the questions of a checklist are often limited to the detection

of defect that belongs to a particular defect types. Since the defect types are based on past defect information [Chernak, 1996], inspector may not focus on defect types, not previously detected, and, therefore may miss whole classes of defects.

Techniques providing more structured and precise reading instruction include both a reading technique known as **'Reading by Stepwise Abstraction'** for code described in [Dyer, 1992], [Linger, 1979], as well as a technique suggested by [Parans, 1987] called **'Active Design Review'** for the inspection of design documents. Reading by Stepwise Abstraction, requires an inspector to read a sequence of statements in the code and to abstract the function these statements compute. Active design review, which is suggested as variation to the conventional inspection methodology, assigns clear responsibility to inspectors of a team, and requires each of them to take active role in an inspection of design artifact.

The gist of **Scenarios-based reading** [Basili, 1997] is the use of the notion of scenarios, which provide custom guidance to inspector, on how to detect the defects. A scenario may be a set of questions or a more detailed description for an inspector on how to perform the document review. Principally, a scenario limits the attention of an inspector, to the detection of particular defects, as defined with the custom guidance. Since each inspector may use a different scenario, and each scenario focuses on different defect types, it is expected that the inspection team together becomes more effective.

The main idea behind, defect-based reading is for different inspectors to focus on different classes, while scrutinizing a requirement document [Portel, 1995]. For each defect class, there is a scenario, consisting of a set of question an inspector has to answer while reading. Answering the question helps an inspector primarily to detect the defect of that particular type.

The rational of **perspective-based technique** is, that there is no single monolithic definition of software quality, and little general agreement about how to define any of the key quality properties such as accuracy, maintainability, and testability. Therefore, inspectors of an inspection team, have to check software quality as well as the software

quality attribute of a software artifact, from different perspectives. For each perspective, one to many scenarios is defined, consisting of repeatable activities, an inspector has to perform, and questions an inspector has to answer. For example, designing test case is a typical activity performed by tester. Therefore, an inspector reading from the perspective of the tester, may have to think about designing test cases to gain an under standing of the software product from the tester point of view [Basili, 1996].

General prescription about which reading technique to use, in which circumstances can rarely be given. However, they can be selected on the basis of following criteria: **Application Context, Usability, Repeatability, Adaptability, Coverage, and Overlap.**

2.2.2.2 The Managerial Dimension of Software Inspection

One of the most important criteria, for choosing a particular inspection approach is the amount of effort, a particular inspection method or refinement requires. Effort is an issue project managers are mainly interested in. Hence this dimension has been referred to as the managerial dimension. To make sound evaluation, that is to determine whether it is worth spending effort for inspection, one must also consider how inspection affect the quality of the software product as well as the cost and the duration of the project in which they are applied.

2.2.2.2.1 Quality:

One of the ways to define quality of software inspection is to measure effectiveness. In order, to be able to compare different inspections, we have to look at the rate at which defects were detected in the inspected artifacts. Therefore, effectiveness can be measured as the density of defect-detected.

Effectiveness = Number of Defects detected / Size.

The other approach, which quantifies the quality of software inspection, is the measure of Defect Removal Efficiency. It is calculated as:

Let D_f be Defect that exist in the document prior to defect detection phase 'f'. The value of D_f is difficult to measure accurately in practice. One approach is to estimate this by the total number of defects found by the time of completion of all defect detection phases (this can be calculated at the end stage of the development cycle).

Let Mf be Defect detected during defect detection defect phase 'f'. Defect removal efficiency E_f of a defect detection phase 'f' across all instances of 'f' is given by:

$$\mathbf{E_f = D_f / M_f} \tag{2.1}$$

2.2.2.2.2 Cost

Harold Green [Crosby, 1978] said it best, "Quality is not only right it is free. And it is not only free, it is the most profitable product we have." The real question is not how much a quality management system will cost, but how much the lack of one will cost.

It is necessary for a project manager to have precise understanding of the cost associated with the inspections. Since inspection is human based activity and therefore, one of the questions sought to be answered throughout: " how much inspection effort is worthwhile?" Thus the effort is the most significant factor in determining the cost of inspection.

Let Q_f be effort spent on finding and fixing a defect in defect detection phase 'f' in Person-hour. M_f is defect detected during the phase detection phase 'f'. Then the, estimated cost C_f of finding and fixing a defect in defect detection phase 'f' across all instances of 'f' is defined as:

$$\mathbf{C_f = Q_f / M_f} \tag{2.2}$$

The estimate can be calculated based on data from other projects or can be calculated a posterior for the project under study.

Different cost models are defined by different companies, associated with their own expert survey, culture and product set with which the company deals with:

Table 2.1 Cost models

Model	Planning Metric
AT&T	50*KSLOC
BEST	SLOC/ (Rate*2)* (Team Size*4+1)
Glib	5.76 * Team size+0.24
HP	SLOC / (Rate*2)*25
BNR	3*KSLOC* 4*8

[www.davidfrico.com/SPC]

2.2.2.2.3 'What to Inspect':

A risk analysis is performed to assess the relative risk associated with each module and it's components. Two factors are more likely to be used in assessing the risk: 1) operational importance to over all system functionality 2) technical difficulty and complexity. The statistical weight is devised for these factors on the scale of 1 to 5(weight can vary from organization to organization depending upon their criteria to determine overall functionality and complexity of the work product).

Over all risk rating for each module is obtained by combining the rating for each factor as:

Overall risk factor = risk factor for operational importance* risk factor for Complexity of module. (2.3)

Depending upon the overall rating. The decision has to make which module to inspect or what fraction of module to inspect [Franz, 1994].

Table 2.1 Cost models

Model	Planning Metric
AT&T	50*KSLOC
BEST	SLOC/ (Rate*2)* (Team Size*4+1)
Glib	5.76 * Team size+0.24
HP	SLOC / (Rate*2)*25
BNR	3*KSLOC* 4*8

[www.davidfrico.com/SPC]

2.2.2.2.3 'What to Inspect':

A risk analysis is performed to assess the relative risk associated with each module and it's components. Two factors are more likely to be used in assessing the risk: 1) operational importance to over all system functionality 2) technical difficulty and complexity. The statistical weight is devised for these factors on the scale of 1 to 5(weight can vary from organization to organization depending upon their criteria to determine overall functionality and complexity of the work product).

Over all risk rating for each module is obtained by combining the rating for each factor as:

Overall risk factor = risk factor for operational importance* risk factor for Complexity of module. (2.3)

Depending upon the overall rating. The decision has to make which module to inspect or what fraction of module to inspect [Franz, 1994].

2.2.2.2.4 Duration:

Inspections not only consume effort, but also have an impact on the product's development cycle time. Inspection activities are required to be scheduled in such a way as all participants involved can participate and fulfill their roles. Thus, the time interval for the completion of all the activities will range from at least a few days to few weeks. Hence, duration might be a critical aspect for a project manager, if time to deliver is a critical issue during development.

2.2.2.3 Assessment Dimension of Software Inspection

When assessing whether inspection provide any benefits, qualitative versus quantitative assessment is needed. While qualitative assessment is often based on the subjective opinion of inspection participants, quantitative assessment is based on data collected in inspection and subsequent defect detection activities, such as testing.

2.2.2.3.1 Qualitative Assessment

Weller [Weller, 1993] states that inspection participation result in a better understanding of the software development process and the developed product. One explanation is that, team members become familiar with the whole system, not just with the part on which each individual is working. Finally, inspection contributes to the social integration [Svendsen, 1992]. Apart from the effects on development team, inspection affects each participant individually. One observation is that inspection participant develop software product more carefully [Doolan, 1992], [Fagan, 1986].

2.2.2.3.2 Quantitative Assessment:

It is based on the data collected in inspection or in the project in which inspection was applied, in terms of cost and quality as described earlier.

2.2.2.4 The Organizational Dimensions of Software Inspection

Fowler [Fowler, 1986] states that the introduction of inspection is more than giving individual a set of skills on how to perform the inspections: it also introduces a new process with an organization. Hence, it effects the whole organization, that is, the team, the project structure and the working environment.

2.2.2.5 The Tool Dimension of Software Dimension:

Currently, few tools supporting inspections are available. Most of them were developed by researchers to investigate software (often source code) inspection and no tool has reached commercial status.

Some of the inspection tools are: (1) PAE (Program Assurance Environment) [Belli and Crisan, 1996] that can be seen as extended debugger and represent an exception in the list of tools. (2) InpecQ [Knight and Myers, 1993] concentrate on the support of the phased inspection process developed by Knight and Myers (3) ICILE [Brother, 1993] support the defect detection phase as well as the defect collection phase in face-to-face meeting. (4) Scrutiny [Gintell, 1995] and CSI [Mashayekhi1993], support synchronous distributed meeting to enable the inspection process for geographically separated development team. (6) CSRS [Johnson, 1997], (7)InspectA [Knight, 1993] (8) ASSIST [Macdonald and Miller, 1995] uses its own process modeling language and executes any desired inspection process model.

All tool provide more or less comfortable document handling facilities for browsing document on-line. However, the use of these tools is limited to a particular development situation and may only lighten the inspection burden.

2.2.3 Inspection Metrics:

The metrics are the most important component of the software inspection. They provide the means for actually observing what is happening in the whole software development process. Inspection metrics are used to monitor not only inspection process, but also the effect of introducing changes to the software development process. Inspection metrics gives a much more reliable picture of what is happening in the software engineering environment then unaided intuition [Glib, 1993].

Inspection metrics are particularly vital because they give insights into areas of requirement planning, architecture, design engineering and documentation which conventional field and test data can not provide with sufficient sensitivities at an early stage.

As discussed earlier, inspection is an inseparable part of the development process. Metrics are intended to convert software inspection into more disciplined engineering technique with stable performance, and for setting objective for process improvements. The difficulty experienced in controlling the quality of the inspection process lies in the fact that inspection is mental activity and cannot be observed directly.

A common problem in developing a metric plan is determination of what to measure. Too often, the solution is to collect every possible measure and figure out what it means, but it places an unnecessary burden on the development staff and adds expenses.

The following set of metrics was collected from [Barnard, 1994] [Grady, 1994b]. These are quite comprehensive to view the complete picture of software inspection process.

1. The size of the document inspected: Line of code for code inspection, number of pages for design/requirement inspection, line of text for test inspection)

2. Number of inspector

3.

a) Preparation rate: Preparation rate is defined as the total inspector preparation time divided by the number of pages, Loc, or whatever is appropriate measure for the situation

Preparation rate = (Size of document inspected/ Preparation time)

b) Inspection rate: Inspection rate is defined as the size of document Inspected per meeting hour

Inspection rate = (Size of document inspected/ Inspection duration)

4. Effort per size: The average hours spent in inspection activities by an inspection team for the size of the document inspected.

Effort per size = (Inspection effort/ Size of document inspected)

Inspection effort = (preparation time + number of participant * Inspection duration rework time)

5. Effort per fault detected: The number of hours spent in inspection activity by the inspection team for a single detected fault.

Effort per fault detected = (Inspection effort/Total fault detected)

6. Fault detected per Size: The number of fault found for every thousand of line of code, or per unit of pages of inspected document.

Fault detected per Size = (Total fault detected / Size)

7. Percentage of reinspection: The percentage of inspection ending with a decision to inspect all or part of the document being inspected again.

Percentage of reinspections = (Number of reinspection dispositions/
Number of inspection)

Where

Number of reinspection disposition = Number of inspection disposition of type reinspect + Number of inspection dispositions of type inspect rework.

8. Defect removal efficiency: The percentage of inspection fault found by inspection

Defect removal efficiency =(Total fault detected during inspection/ Total fault detected)

9. Maturity of inspection process: We can estimate maturity of inspection process in a division in terms of the <u>extent of inspection adoption</u>, and the maturity of inspection in a company, as an average of these figures. Maturity of inspection is a new metric. Gardy [Grady, 1994b] define the equation as:

Extent of adoption =(inspection-process maturity*(percentage of Project using inspections) + Weighted percentage of document inspected)*(constant))

Inspection process maturity is a constant from 1 to 14, based on a five level model in which level 1 and 2 are informal peer reviews of formal walk through (weight 1and 3), level 3 is an industry-typical inspection process (weight 10), level 4 is a best practice inspection process (weight 12), and level 5 links formal inspection to a defect prevention activity (weight 14).

Percentage of projects using inspection: The number of inspection in a project should be four or more for inclusion in this percentage.

Weighted percentage of document inspected: Different types of defects require different relative cost fixing them.

Constant: is used to fit the extent-of-adoption metric into a range of 0 to 100.

10. Grade of competence: It explains the skill of the inspector in terms of knowledge gap or the quality of the inspector's comment. The knowledge gap is referred as the difference between the inspector's and designer knowledge. This can be represented in terms of the number of years of experience or the number of case worked or in the problem domain. This evaluation allows the inspector leader to analyze the competence of the inspector and decide if any training needed for the inspectors.

11. Complexity of the code: Measured with Halstead's volume (V) or McCabe's complexity (CC) value.

2.2.3.1 Suggested value for metrics:

Depending on the objective of the concerned measurement, the characteristic of the metrices is to be selected. It is natural that the metrics for inspection quality status will be most essential when monitoring the inspection process, and thus summary of these is presented in Table2.2. The suggested values are derived from literature and experience in industry, which are not in general statistically reliable. One should also be wary of variations derived from individual human factor, interruption and the number of problem encountered. Jones [Jones, 1991], for example warns that variation in inspection preparation rate and inspection rate can deviate from the estimate by + or- 50 percentage.

The size of the requirements and design document may be up to 10 pages per document size [Grady, 1994a]. When using the LOC metric, the suggested size of artifact to be inspected should be less than 500LOC [Barnard, 1994] or less than 300 LOC. The cyclomatic number of a function should be less than 100 and that of a module less than 100, were as Halstead's volume value for a function should be less than 100 and that for a module less than 8000 [Pressman, 1995].

Table 2.2 Suggested values of matrices

Average LOC inspected	< 40/500 LOC
Other complexity metrics	CC< 15/100, V< 100/8000, Function/module
Average preparation rate	< 150- 200 LOC
Average inspection rate	<150-200 LOC
Preparation /inspection time ratio	1.25-1.4
Average effort/KLOC	
Average effort/fault detected	50 hours/KLOC
Total fault detected/KLOC	0.5 –1 hour/fault
Reinspection or exit	40faults/KLOC
Defect-removal efficiencies	0.25major/pages
Extent-of-adoption	74%, 61%, 55%
	26

The average preparation rate and inspection rate is largely accepted to be less than 150-200 LOC per hour for code and less than 250-400 LOT per hour for design documents, although the ratio at time is thought to vary in the range of 1.25 to 1.4[Gardy, 1994a]. Jones suggested 25 pages/hour for preparation and 12pages/hour for meeting in the case of requirements, 45 pages/hour and 15 pages/hour for functional specification, 50pages/hour and 20pages /hour for logic specification, 150LOC/hour and 75 LOC/hour for source code and 35 pages/hour and 20 pages/hour for user documents [Jones, 1991].
The defect removal efficiencies estimates are between 50-90% or 74% for HLD, 61% for LLD and 55% for Code [Glib, 1993]. For total inspection effort, [Barnard, 1994] suggests, that 50-hour should be spent per KLOC as an average effort unless data are available from previous projects.

[Ackerman, 1989] and [Russell 1991] suggested average fault detected per KLOC be in range of 40-50. [Glib, 1993], gives a guideline for reinspection/ exit decisions, according to which an exit is suggested if there are less than 0.25 major defects per page remaining. A necessary criterion for this decision is that inspection and preparation rate did not exceed optimum rate (150-200 LOC/hour). Grady [Grady, 1994b] use the value 26 as acceptable limit for the extent-of-adoption metrics.

2.2.4 Variants of Software Inspection Process.

2.2.4.1 N-Fold Inspection

Martin et al. proposed the *N-fold inspection method* [Martin, 1990]. This inspection method is based on the hypotheses that a single inspection team can find only a fraction of the defects in a software product and that multiple teams will not significantly duplicate each other's efforts. In an N-fold inspection, N teams carry out parallel independent inspections of the same software artifact. In a sense, N-fold inspection scales up some ideas of scenario-based reading techniques, which are applied in the conventional inspection approach on an individual level, to a team level. The inspection participants of each independent inspection follow the various inspection steps of a conventional inspection as outlined in previous section, that is, individual defect detection with an Ad-hoc reading technique and defect collection in a meeting. The N-Fold inspection approach ends with a final step in which the results of each inspection team are merged into one defect list.

2.2.4.2 Active Design Review

Parnas [Parnas, 1987] suggest an inspection method denoted as *Active Design Reviews (ADR)* for inspecting design documents. The authors believe that in conventional design inspection inspectors are given too much information to examine, and that they must participate in large meetings, which allow for limited interaction between inspectors and author. To tackle theses issues inspectors are chosen based on their specific level of expertise skills and assigned to ensure thorough coverage of design documents. Only two

roles are defined within the ADR process. An inspector has the expected responsibility of finding defects, while the designer is the author of the design being scrutinized.

ADR process consists of three steps. It begins with an overview step, where the designer presents an overview of the design and meeting times are set. The next step is the defect detection step for which the author provides questionnaires to guide the inspectors. The questions are designed such that they can only be answered by careful study of the design document, that is, inspectors have to elaborate the answer instead of stating yes/no. Some of the questions reinforce an active inspection role by making assertions about design decisions. For example, he or she may be asked to write a program segment to implement a particular design in a low-level design document being inspected. The final step is defect collection, which is performed in inspection meetings. However, each inspection meeting is broken up into several smaller, specialized meetings, each of which concentrates on one quality property of the artifact. An example is checking consistency between assumptions and functions, that is, determining whether assumptions are consistent and detailed enough to ensure that functions can be correctly implemented and used.

Active Design Review is an important inspection variation because ADR inspectors are guided by a series of questions posed by the author(s) of the design in order to encourage a thorough defect detection step. Thus, inspectors get reading support when scrutinizing a design document. Although little empirical evidence shows the effectiveness of this approach, other researchers based their inspection variations upon these ideas [Knight, 1991].

2.2.4.3 Phased Inspection

Knight and Myers [Knight, 1991] [Knight, 1993] suggested, the Phased Inspection method. The main idea behind Phased inspection is for each inspection phase to be divided into several mini inspections or phases. Mini-inspections are conducted by one or more inspectors and are aimed at detecting defects of one particular class or type. This is the most important difference to "Conventional" inspections, which check for many

classes or, types of defects in a single examination. If there is more than one inspector, they will meet just to reconcile their defect list. The phases are done in sequence, that is, inspection does not progress to the next phase until rework has been completed on the previous phase.

Although Knight and Myers state that phased inspections are intended to be used on any work product they only present some empirical evidence of the effectiveness of this approach for the code inspections.

However, Porter [Porter, 1997b] argue based on the results of their experiments that multiple session inspections, that is, mini-inspections, with repair in between are not more effective for defect detection but are more costly than conventional inspections. This may be one explanation why we did not find extensive use of the phased inspection approach in practice.

2.2.4.4 Verification-based Inspection

Verification-based Inspection is an inspection variation used in conjunction with the Cleanroom software development method. Although this method requires the author(s) to perform various inspections of work products, the inspection process itself is not well described in the literature. It consists of at least one step, in which individual inspectors examine the work product using a reading technique denoted as "Reading by Stepwise Abstraction". This reading technique is limited to code artifact, though it provides a more formal approach for inspectors to check the functional correctness [Dyer, 1992b]. Little information on the inspection process after the individual defect detection step was found. However, the Cleanroom approach is a one of the few development approach in which defect detection and inspection activities are tightly integrated in and coupled with development activities.

CHAPTER THREE.

Software Quality Models

There does not exist a standard method for determining the impact of specific variable connected to software inspection process on software development cost and quality.
In the literature four approaches to this problem have been considered: Conducting a Controlled Experiment, Employing Pilot Projects, Using traditional Quality Management Model, and Developing System Dynamic Model.

3.1 Controlled Experiment

An ideal way to evaluate process improvements is to conduct a controlled experiment in which all factors, except the independent variable (e.g.: - team size, team mix, etc.) are kept constant. Although much can be learned though this approach about the general effectiveness of software inspection process improvements, it is normally not practical to experiment over the entire development life cycle of a project of significant size. Thus, it is often impossible to assess the impact of proposed improvement on software development cycle time.

3.2 Pilot Projects

Another approach typically followed is the utilization of some sort of "pilot projects" to assess the new technology. Although considerably weaker than controlled experimentation, the pilot studies often reveal the great merit of the proposed technology. Guidelines have been proposed to improve the effectiveness of pilot projects [Barbar, 1995], but it remains difficult to assess the unique impact of the proposed improvement technology on the software inspection process, due to the interaction of other project variables and to generalize the result of other projects.

3.3 Quality Management Models

It is important to assess the quality of the software system when development work is complete. Software quality management models serves as measurement tools. Early warning signs or improvements can be detected, and timely management actions can be taken. Any software quality management model must cover the early development stages in order to predict quality in the later stages.
Some of these models are discussed in next subsections.

3.3.1 The Rayleigh Model [Kan, 1991]

A Rayleigh model is specific case of the Weibull distribution when m=2. Its CDF and PDF are:

CDF: $F(t) = 1 - \exp(-t/c)^2$ (3.1)

PDF: $f(t) = 2/t^* (t/c)^{2*} \exp(-t/c)^2$ (3.2)

The Rayleigh PDF first increases to a peak and then decreases at a decelerating rate. It has been well established that software projects follow a life-cyclic pattern described by the Reyleigh density curve [Putnam, 1978]. The basic assumption for using the Rayleigh model is that, if more defects are discovered and removed in the early development phase, fewer will remain in the remain phase, which result in better quality of the system.

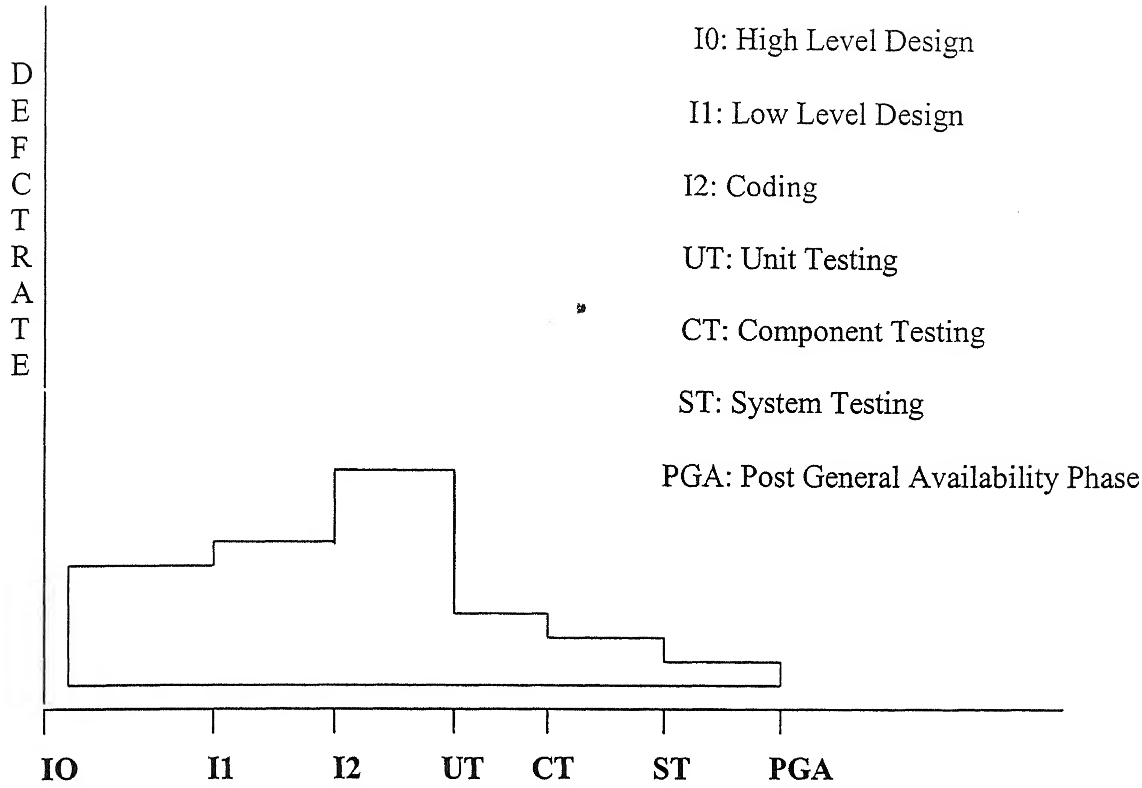


Figure. 3.1 Rayleigh Model [Kan, 1991]

Figure 3.1 depicts the defect removal pattern in the development stages: high-level design (I0), low level design (I1), coding (I2), unit test (UT), component test (CT), and system test (ST). The defect rate of the last phase in the figure, post general availability phase (PGA), is the target of the estimate of Rayleigh model.

3.3.2 Goel –Okumoto Non Homogeneous Poisson process models [Goel, 1985]:

This model assumes that a software system is subjected to failure at random times caused by fault present in the system. It is one of the simplest models, and extensively used in industry. It assumes Poisson distributions for number of failure in a specified time ‘t’.

It propose following form of model

$$P\{N(t) = y\} = ((m(t))^{y}/y!)\, e^{-m(t)} \qquad y=0,1,2 \ldots\ldots.. \tag{3.3}$$

Where

$m(t)$ = expected number of failure by time 't'.

$$m(t) = a(1-e^{-bt}) \tag{3.3.1}$$

Where

a = expected number of failure to be eventually observed.

b = fault detection rate per fault.

N (t) = Represents cumulative number of failure up to time 't' i.e. y

This model is also known as the exponential model [Misra, 1983]. In addition to the standard exponential model, two variants models that were proposed by Ohba [Ohba, 1984], for software reliability analysis and also in use: The delayed S model and the inflection S model. The Goel –Okumoto model has no inflection points, the delayed S model has a slight inflection at the beginning of the curve, and the inflection S model has substantial inflection at latter part of the curve. The three models as well as the other similar model are referred to as reliability growth models [Ohba, 1984].

A weakness with these models is that they require substantial data and are able to predict results only at the end of the development process.

3.3.3 The Discrete Defect-Removal Model.

Whereas the Reyleigh curve model describes the pattern of defect removal, the discrete defect removal model (DRM) deals with both defect removal and defect injection. The DRM takes a set of error-injection by development steps and a set of testing and inspection effectiveness rate as input, then models the defect removal pattern step-by-step. The defect removal and defect injection at each step are described in figure 3.2.

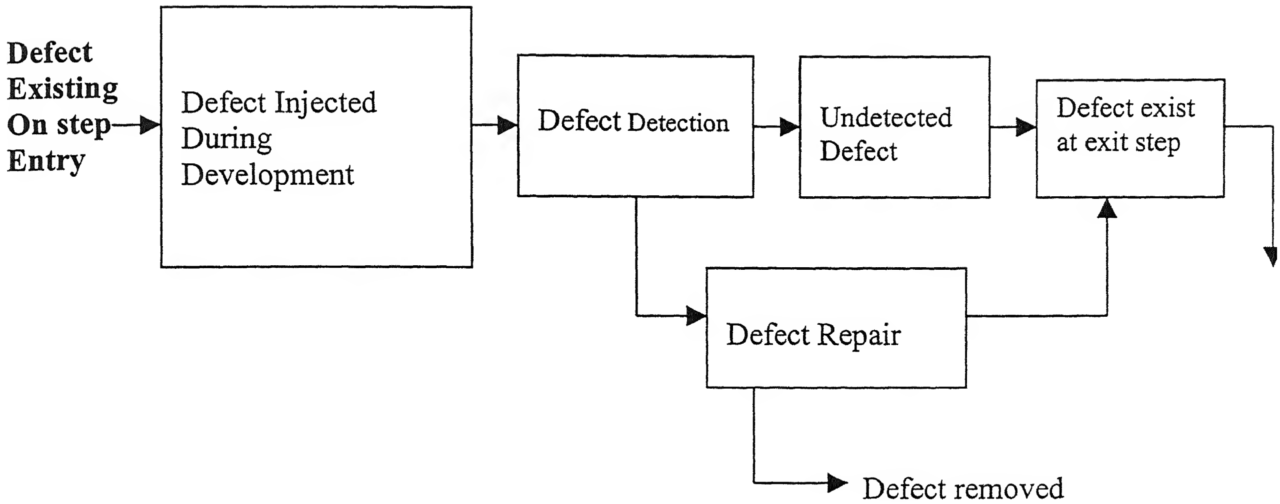


Figure. 3.2 Defect Removal and Injection Steps [Kan, 1991]

Defect at the Exit of a Development Step	=	Defect escaped from previous Step	+	Defect injected in Current Step	-	Defect removed in Current Step

Figure. 3.3 Simplified view of Defect Removal and Injection Step [Kan, 1991]

The error injection rates and the inspection effectiveness are usually based on estimates from the previous release or from historical data. If the total number of defects, or defect rate, of the system are known, the final quality index can be calculated based on the model output at the last development step.

However unlike the other parametric model, the DRM cannot estimate the final quality index. It cannot do so because, the total defect rate (or, for that matter, the latent defect rate when the system is shipped), the very target for estimation, is needed as input of the model. It is a tracking tool instead of a projection tool. The rationale behind this model is that if one can ensure that the defect-removal pattern by step is similar to that for a

previous experience, one might reasonably expect to see approximately the same final quality index.

3.3.4 The PTR Sub Model

Where the DRM, is used for tracking quality status during development, the other model used during the machine testing stages is the Program Trouble Report (PTR). The PTR is the vehicle for defect reporting and integrating fixes when the code is placed under formal change control process. Valid PTRs are, therefore code defects found during testing. A PTR sub model is necessary because the period of formal machine testing (part of unit test, integration test, and system test) usually spans months, and it has to be ensured that the chronological pattern of defect removal is also on the track. Simply put, the sub models spreads over time the number of defects that are expected to be removed during the machine testing phase, so the more precise tracking is possible. It is a function of three variables:

- Expected overall PTR rate
- Planned or actual lines of code integrated over time
- PTR surfacing pattern after code is integrated
- The expected overall PTR rate can be estimated from historical data. Lines-of-code integration over time is usually available in the current implementation plan.

The biggest weakness of this model is that it can be used only at the testing stage of the software development process.

The largest obstacle with software quality management model is data constraints. Except the DRM model, all other model in this category requires substantial data to find out the parameter of the models. With better data, these models will provide better explanation and more accurate projections of the complex reality. The only exception in this category is DRM model.

3.4 System Dynamics Model

System dynamics techniques have recently been used to model "high-level" process improvements corresponding to SEI levels of maturity [Howard, 1994]. System dynamics models differ from traditional cost estimation models and executable process models, in that, they model both the software development process and the behavioral aspects of software development.

The following description of the System Dynamic model is taken from Iona Rus [Iona Rus, 1996].

System dynamics is defined as the "the application of feedback control systems principles and technique to modeling analyzing, and understanding the dynamic behavior of complex systems [Abdel, 1989]". This technique was applied for the first time to the software development process by Abdel-Hamid and Stuart Madnic [Abdel, 1989]. System dynamics modeling is primarily based on cause and effect relationships. These cause and effect constantly interact, while the software development process model is being executed, simulating the dynamic interaction of the system.

A system dynamics model can contain relationships between people, product, and process in software development organization. The most powerful feature of system dynamics modeling is realized, when multiple cause-effect relationships are connected forming a circular relationship, known as a feedback loop. The concept of a feedback loop reveals that, any actor in a system will eventually be affected by its own action. The following scenario can illustrate a simple example of the ideas of cause-effect relationships and feedback loops affecting people, product, and process:
Consider the situation in which developers, perceiving their product is behind schedule (cause), modify their development process by performing fewer quality assurance activities (effect/cause), leading to a product of lower quality (effect/cause), but giving the temporary perception that the product is back on schedule.

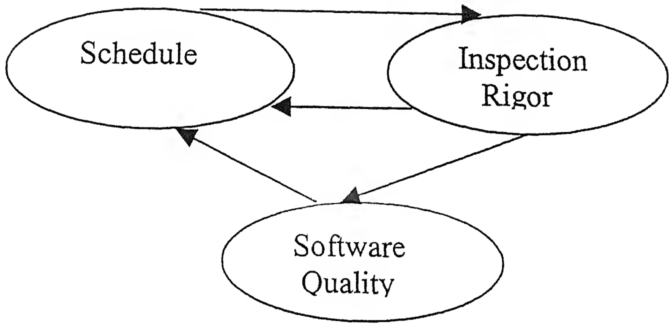


Figure. 3.4 Cause-effect relationships and feedback loop example [Abdel, 1989]

The developers perceived the product to be behind schedule, took action, and finally, perceived the product to be back on schedule, as a result of their actions. Secondary effects due to the developers' actions, however, such as the lower quality of the product, will also eventually impact the perception the developers have, with regard to being on schedule, and will necessitate further actions being performed. These cause-effect relationships are explicitly modeled using system dynamics techniques.

Because system dynamics models incorporate the ways in which people, product, and process react to various situations, the models must be tuned to the environment that they are modeling. The above scenario of developers reacting to a product that is behind schedule would not be handled the same way in all organizations. In fact, different organizations will have different levels of productivity, due to the experience level of the people working for them and the difficulty of the product being developed. Therefore, it is unrealistic for one model to accurately reflect all software development organizations, or even all projects within a single development organization.

Once a system dynamics model has been created and tailored to the specific development environment, it can be used to find ways to better manage the process to eliminate bottlenecks and reduce cycle time. Currently, the state-of-the-practice of software development is immature. Even immature software development environments, however, can benefit from this technique. The development of a model forces

organizations to define their process and aids in identifying metrics to be collected. Furthermore, the metrics, and the model that uses them, do not have to be exact in order to be useful in decision-making [Stark, 1994]. The model, and its use, will result in a better understanding of the cause-effect relationships that underlie the development of software. The power of modeling software development, using system dynamics techniques, is its ability to take into account a number of factors that affect cycle time to determine the global impact of their interactions, which would be quite difficult to ascertain without a simulation. The model may be converted to a management simulator, providing researchers and software project managers with the ability to perform controlled experiments on their development environment.

Much of the research in system dynamics modeling, relating to software engineering, has been done at M.I.T. System dynamics modeling was developed in the late 1950's at M.I.T. The first scholarly effort to apply this technique to project management within a research and development environment was done by Roberts in his doctoral dissertation, "The Dynamics of Research and Development" [Roberts, 1964]. His model simulates the full life cycle of a research and development project. His work is mentioned due to the great similarity between research and development projects and software development projects [Gehring, 1977].

His students at M.I.T later extended Roberts' work. They modeled multi-project environments [Kellner, 1991], allocation of manpower resources, development groups [Richardson, 1982], and rework [Roberts, 1981].

These models were the basis for Abdel-Hamid's work [Abdel, 1989]. He specifically modeled the software development environment. Abdel-Hamid tailored his model to the software development process at the Systems Development section of NASA's Goddard Space Flight Center.

Recently, more positive results have been published regarding the use of system dynamics techniques in modeling aspects of the software development process

[Chichakly, 1993], [Copper, 1993], [Lin, 1993], [Howard, 1995] These results strengthen the claims made by Abdel-Hamid, regarding the usefulness of system dynamics modeling, with respect to software development.

3.5 Field Study

3.5.1 Introduction

To study the practices followed by the software industry in India, it was decided to do a field study, on some representative units of the industry. Due to constraints of time and resources, the practices of only one unit situated at, Lucknow was studied. This unit is a part of India's largest software firm with a turnover of Rs.2485 crore. The unit was chosen as representative of the practices of the Indian software industry, especially the organized sector. This unit of organization generally involve in following kind of project:

- Development Projects
- Product Support/Implementation Projects

3.5.2 Methodology

The methodology followed for the field study was to conduct a series of semi-structured interviews with the people working in the SEPG (Software Engineering Process Group) of the organization. A few open-ended questions were posed to the managers and executives of the organizations. The replies were noted by hand and the executed were prompted gently, to keep them on the topic of inspection practices, when they strayed.

The interviews were then analyzed using content analysis. A number of categories were formed, and then the results were collated, to map the process being followed by the company.

3.5.3 Summary of Results

The inspection process is divided in two categories:

1. Internal Inspection
2. External inspection

The internal inspection goes like this: for example there are two developers employed in the same project, the work product developed by one developer, is inspected by the other developer, the other developer after inspecting it, provides feedback to the author of the work product concerning, the defects he has detected. The author will rectify the identified defects and pass it for external inspection.

The external inspection is done by SEPG (Software Engineering Process Group). The method they follow to inspect the work product matches with what has been prescribed by Fagan [Fagan, 1976]. The inspection process includes the following steps: planning, overview, preparation and, inspection meeting, reworks and follow-up. These steps are described in detail in chapter 2. In some cases as per the requirement, ignoring some of the steps, like meeting or overview, follows the modified Fagan inspections method.

3.6 Selected Model

By investigating the strength and weakness of different models as described in previous section the decision was made to model software inspection process by using the concept of DRM model and incorporating in the feature of system dynamic modeling approach. As it gives the DRM model the flexibility to vary its input parameter with respect to people product and process related to specific development environment.

The DRM is useful management tool, which can track current quality status during development and verify the final quality index that is estimated from independent sources.

More importantly by incorporating the feature of system dynamics modeling in the DRM model we can evaluate the influence of variable connected to software inspection process on the software quality index and on the cost of inspection process.

CHAPTER FOUR

Designing a Learning Tool for Management of Software Inspection Process (A Management Game)

4.1 Introduction to Management Game

Management games are "games" in the sense that there are participant, a set of rules and a method of scoring. There is a specific branch of mathematics called "The Theory of Games" which was founded by Von Neumann and Morgenstern. The object of game theory is to analyze games and to arrive at the best methods for playing them. When executives play a management game, they tend to learn from the experience gained, from trial and error type decisions and the feed back from the simulation exercise. At the same time some of these situations can be analyzed using theory of games to arrive analytically, at the best strategy that should be followed. Thus the term "Management Game" is used to refer primarily to those simulations in which groups of human beings are engaged in taking decisions, usually for **training purpose** [Kibbee, 1962].

The player of management game, usually are briefed, that they have just taken over the management of company in an industry with which they may not be acquainted, that they will have to learn from experience and utilize their learning to apply good management principles.

In management games, the concentration on decision-making produces, many of the same advantages, as do case studies. However management games add two extremely important elements to the case studies approach, the objectivity of the feedback and use of the time dimension. Objective feedback is provided in a management game by set of programmed relationships, which transform the decisions (input) taken in a specified frame of time, into performance reports (evaluations).

4.2 Designing a Game

Designing a game for the software inspection process continually involves compromises, deciding what to include as well as what to leave out. A game is a model of some segment of reality, and modeling implies abstraction, the inclusion of relevant items and omission of irrelevant items.

4.2.1 The four Constraints

The four basic constraints in designing the game are:

- Objective/Purpose
- Simplicity
- Verisimilitude
- Reality

4.2.1.1 Objective/Purpose

Objective

To develop a management game of the software inspection process, in order to assess the impact of different variables connected with the software inspection process, on the quality of the end product.

Purpose

To develop a 'tool' to train prospective software project manager, to understand the impact of different variables, and to learn to use required tactics by simulated experimentation.

4.2.1.2 Simplicity

The need for simplicity can be an extremely important constraint on the model and can often make it difficult to satisfy all the requirement of purpose. There are at least three

facets to simplicity, and they can often be in conflict with each other. These are: simplicity of participation; simplicity of abstraction; and simplicity of administration.

4.2.1.3 Verisimilitude and Realism

Verisimilitude is the appearance of the reality to the player, but this does not imply necessarily, the realism of the model. To sustain involvement, the player must not be distracted by too obvious an artificiality of the model. However, as is well known in the theatre, an illusion of reality can some times be more convincing then the reality itself [Kibbee, 1962].

The degree of the realism needed in the model depends on the training objectives. The purpose of our game is to exemplify certain management skills such as analysis, planning, etc., and as such the exact relationships between checking rate and defect detected or experience level with defect detected is not important.

A common game objective is, that the players learns to benefit from experience, and thus are able to deduce relationship between cost and quality (defect detected). If the objectives of the game are to teach exact known characteristics of software inspection process of a particular organization for a particular project, then a realistic equation-representing defect detected with respect to other variable is needed. Similar remarks apply to other element of the model.

4.2.2 Methodology

The process involves three information-gathering steps:

The first was to conduct an extensive review of the literature as described in chapter 2. This review helps in filling many knowledge gaps, giving rise to a base of software inspection process.

In the second steps a field study was conducted in a leading software development organization, where we conducted a series of informal interviews with member of the SEPG (Software Engineering Process Group). The purpose of this set of interviews was to provide us with a first hand account of, how software quality assurance actually takes place in software development organizations.

We had also taken suggestion from Mr. Vivek Bhagwat who is having eight-year experience in software development organization. After that the relevant factor for the model were identified.

4.3 Model Development

System models can be classified into physical and mathematical, static and dynamic, numerical and analytical [Gordon, 1978]. The model that is presented here is mathematical, dynamic, numerical, and executable, allowing process simulation.

Developing the model of the software inspection process involves, identification of entities, factor, variable, interactions and operations that are present in that process. This is large task that needs to be broken down, to grasp it into meaningful way.

The entities include mainly people (inspectors, developers etc.), and also include things such as facilities, computer equipment software tools, documentation, and work instructions.

Factor relevant to general software inspection process include:

- Product factor: some of the possible variable affecting the number of defect in the product, include, size of the document, author experience level, time period when it was written, and functionality to be developed.
- Inspectors' factors: some of the possible variable affecting the number of defect detected include inspector experience level, time spent on detecting the defect etc.

(Note that we have only considered the effect on the number of defect detected in individual checking i.e. in preparation stage only).

- Team factor: Different inspectors have different ability and experience and possibly interact differently, with each other. Hence the team compositions are the most important variable with respect to this factor.

The factors and variables are things that are not difficult to identify. The modeling complexity increases when identifying operations and interactions.

In order to identify the interactions among the relevant factors and variables in the software inspection process cause and effect describing the relationships between them is needed. In the following section we discuss different causal models explaining the relationships relevant to situation under consideration.

4.3.1 A Causal Model for Explaining Inspection Quality

A high quality inspection must ensure that most of the detectable defects in a software product are, indeed, detected. Therefore, we are interested in the factors that have an impact on the number of defect detected.

The principal factors are the team characteristics, effort, the reading technique, the organization of the defect detection activity, and product characteristics. The major team characteristics are, the number of inspector and their experience. The major product characteristics are the type of the product that is inspected, the difficulty of the product, such as its complexity, the size of the product and its initial quality

These principal factors impact the number of defect detected in the following manner:

- Increasing the number of inspectors is expected to increase the number of defect detected in an inspection.
- Using very experienced inspector is expected to increase the number of detected defect in an inspection. This stem from the fact, that if inspector is well versed

with the application domain, he already knows many potential pitfall and problem spots.

- Spending more efforts for defect detection is expected to increase the number of defect detected in inspection.
- The difficulty of a product is related to the defect-proneness. This means that a more difficult software product contains more defects.

The factors "Reading Technique" and "Organization of the defect detection activity" are not easy to quantify, hence no proper input and output relationship inference can be drawn. More ever, we have to mention that other factors, such as tool support, may have an impact as well. Nevertheless, these factors need to be investigated and, we did not include them here because we focused on most prevalent factor, which determine the input to inspection process.

4.3.2 A Causal Model Explaining Inspection Effort

The principal factors determining inspection effort are team and product characteristics. The principal factors impact inspection effort in the following manner:

- Increasing the number of people increases the inspection effort
- The more experienced the inspector, the less effort they consume for defect detection and, thus, for the overall inspection.
- The more difficult the product the more effort is required for inspecting it.
- The larger the size of the inspection product, more effort is required for its inspection

4.4 Model Structure

Since we have planned the management game for the software inspection process, we have decided to choose two important factors for evaluation; from managerial dimensions of software inspection process i.e. the **cost and the quality.**

We have selected the inspection process model as suggested by Fagan [Fagan, 1976] as our base model and integrated it with a typical organization, waterfall software development model.

The figure 4.1 represents process steps and effort involved in inspecting the work products. However, it does not represent time and manpower allocations to perform each step in the inspection process so as to keep the diagram and idea represented simple.

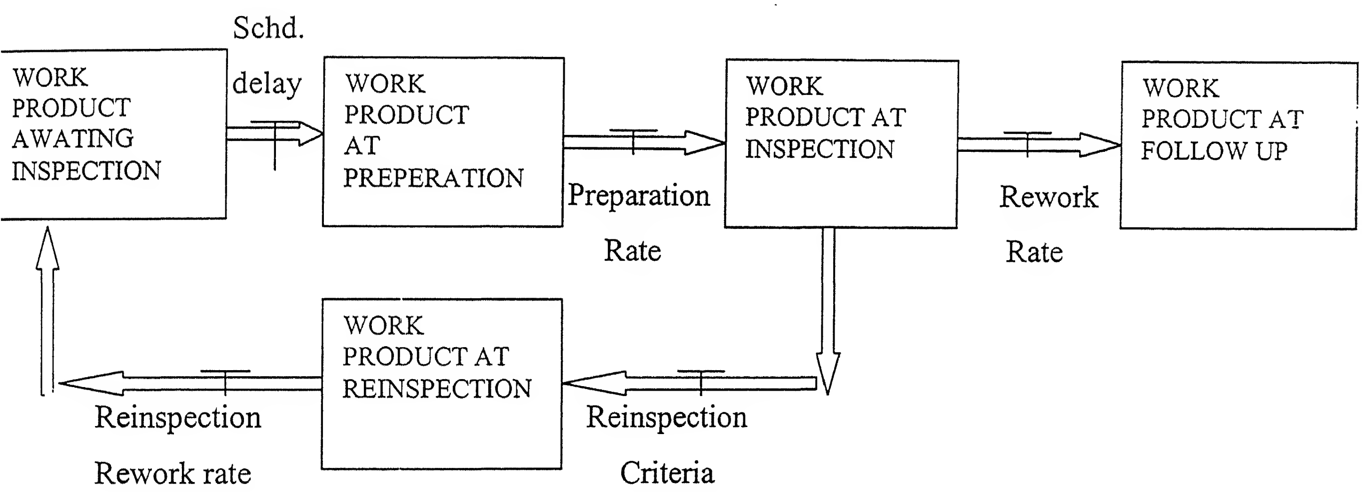


Figure 4.1. Simplified Steps of Inspection Process

As already stated, that one of the most important aspect of System Dynamic Modeling are the use of feedback loop showing the cause and effect relationships. The feedback loop shown in figure 4.2 shows, how the factor expressed in causal model of software inspection process behaves dynamically.

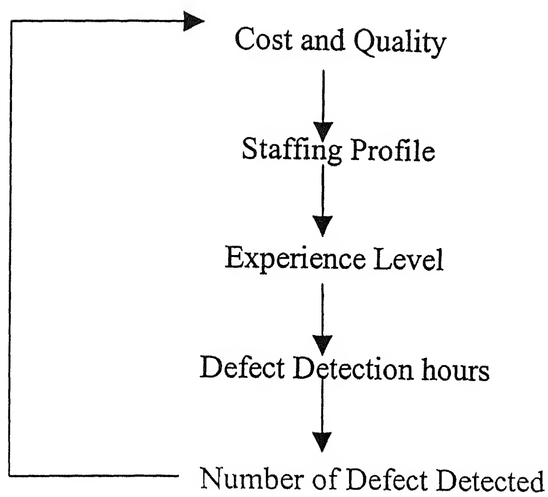


Figure 4.2 Feedback loop of Inspection Process

The feedback loop shown in figure 4.2 takes into consideration the amount of defect detection hours, which translate into number of defect detected, which in turn reflect the quality level achieved by inspection process. An inspector with less domain experience might spent more time on defect detection and will detect less defect as compared to inspector with high domain experience and hence more defect will escape to the next phase resulting in poor quality index and may result in at lower or higher cost value depending upon the cost of the inspector.

Figure 4.3 shows, a representative implementation of the interface between DRM model and variables connected to software inspection process. Feedback loop described in figure 4.2 provides the basis for varying the input parameter of the model.

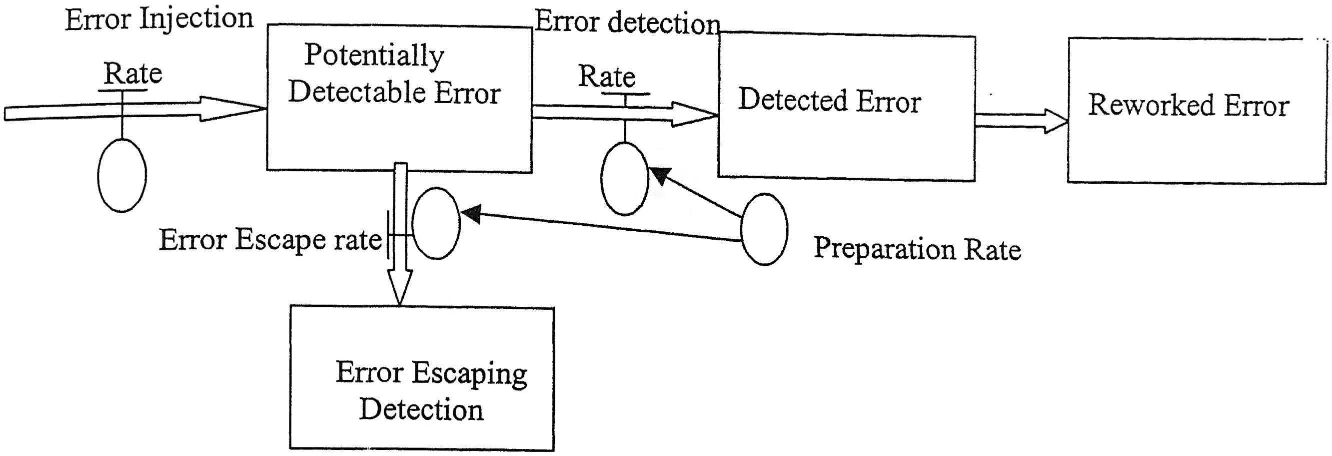


Figure 4.3 Impact of Inspection on error.

Figure 4.3 also represent modeling of errors in the DRM model and illustrate the impact inspection has on the error detection in the DRM model.

When defect get escaped to the next stage, its magnitude gets magnified. This fact is taken into consideration by, incorporating the magnitude amplification factor for defects escaping in each phase. The concept is shown in the figure 4.4

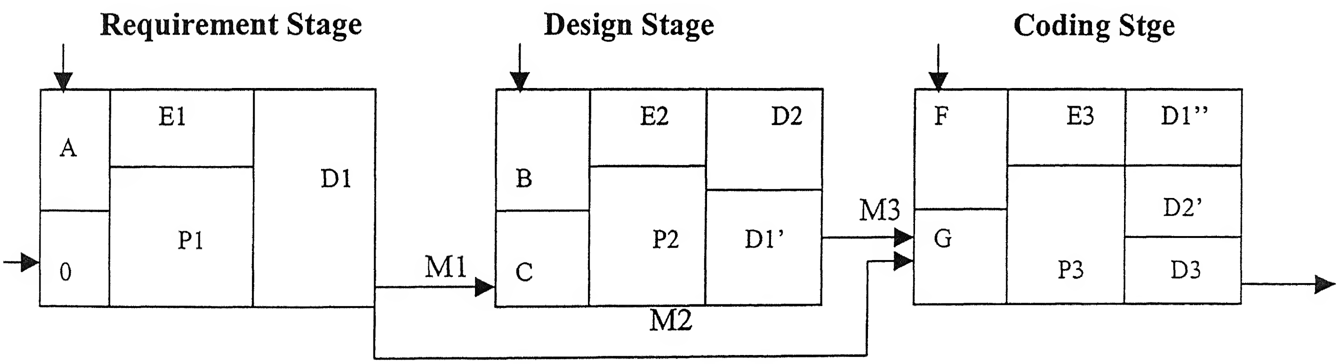


Figure 4.4 Modeling the escaped defects

LEGENDS

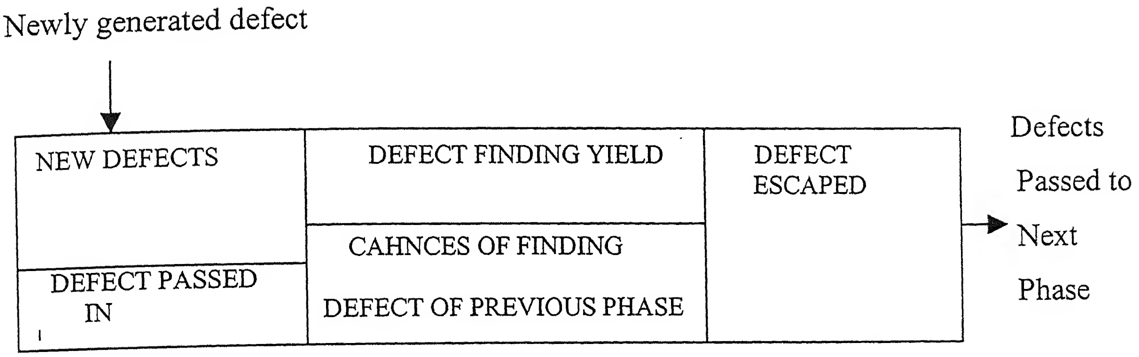


Figure 4.5 Legends

The model is general enough, as it requires no structural change to handle other variants of conventional Fagan inspection process. The following parameters are required for calibration of the model:

- Inspection Effectiveness (of inspector at each experience level)
- Defect injection rate for various phases of software inspection process
- Average requirement, design defect amplification
- Chances of detection of defect escaped from previous phase.

The above-discussed parameters generally will vary from organization to organization and also from project to project with in an organization.

4.5 Mathematical Model

In this section a mathematical model relating cost of inspection with inspection effort is described.

Let

DI_i = Defect injection rate of phase 'i'

ED_i = Expected number of defects in the work product at phase 'i'

S_{ijp} = Size of the work product at phase 'i' to be checked by the p^{th} person at the j^{th} week

EF_{ijp} = Effectiveness of p^{th} person for the i^{th} phase at the jth week

ND_i = Number of defect detected at phase 'i'

Where

j = 1,2,3.... K_i the number of week for phase 'i'

p = 1,2 ,3 .... N_i the number of persons available and used for phase 'i'

C_i = Defects escaped from the previous phase i.e. i to the phase i+1

CR_{ijp} = checking rate of the p^{th} inspector at i^{th} phase on j^{th} week

CH_{ip} = Checking Hour of p^{th} inspector of the i^{th} phase

V_i = Cost of Inspection Process at the ith phase

M1, M2, M3 are the defect amplification factor

M1= Requirement defect amplification factor which escaped to design stage

M2= Design defect amplification factor which escaped to coding stage

M3 = Requirement defect amplification factor which escaped to coding stage

P1, P2, P3 are the chances of detecting the fault which are coming from previous phase in the current phase

P1= Chances of requirement defect detection in the design phase.

P2= Chances of requirement defect detection in the coding phase.

P3 = Chances of design defects detection in the coding phase.

Then

$$ND_i = \sum_{p=1}^{p=N_i} DI_i * (\sum_{j=1}^{j=N_i} (S_{ijp} * EF_{ijp})) + E_i \qquad (4.1)$$

Where

For requirement phase

$E_1 = 0$

For design Phase

$E_2 = M1 * P1 * C_{i-1}$

Where C_{i-1} = Defects coming from the requirements phase.

For coding phase

$E_3 = M2*P2\ (1-P1)*M1*C_{i-1} + M3*P3*C_{i-2}$

Where C_{i-2} = Defects coming from the requirements phase, C_{i-1} = defects coming from the design phase.

$$C_i = \sum_{p=1}^{p=N_i} DI_i * (\sum_{j=1}^{j=K_i} (S_{ijp} * (1 - EF_{ijp}))) + Li \quad (4.2)$$

Where

For requirement phase

$L_1 = 0$

For design Phase

$L_2 = (1\text{-}P1)*MI*C_i\text{-}_1$

Where C_{i-1} = Defects coming from the requirements phase.

For coding phase

$L_3 = M2*(1\text{-}P2)((1\text{-}P1)*M1*C_{i-2})) + M3* (1\text{-}P3)*C_{i-1}$,

Where C_{i-2} = Defects coming from the requirements phase, C_{i-1} = defects coming from the design phase.

$$CH_{ip} = \sum_{p=1}^{p=N_i} \sum_{j=1}^{j=K_i} (S_{ijp} / CR_{ijp}) \quad (4.3)$$

$$V_i = \sum_{p=1}^{p=N_i} (CH_{ip} * C_p), \quad (4.4)$$

Where C_p is the per hour cost of the p^{th} inspector

4.6 Source of Data

In this section, the inspection data that matches the model input variables were identified from the published software engineering literature. It is expected that in the literature data is sourced from companies that had some success in implementing the inspections process effectively.

4.6.1. Effectiveness of Defect Detection Phases

Fagan [Fagan, 1976] presents data from a development project at Aetna life and casualty, Harford, Connecticut, USA. An application program of 8 module (4439 non commentary source statement) was written in Cobol by two Programmers. Design and code inspection were introduced into the development process. The number of inspection participant ranged between three to five. After six month of actual usage, 46 defects were detected during development and usage of the program. Fagan reports, that 38 defects that have been detected by design and code inspection together, yielding defect detection effectiveness for inspection of 82%, and the remaining 8 defects were found during unit test and preparation for acceptance test.

In another article, Fagan [Fagan, 1986] published data from a project at IBM Rochester, United Kingdom. A program of 6271 LOC in PL/1 was developed with seven developers. Over the life cycles of the project, 93% of all defects were detected by inspections.
Weller [Weller, 1993] presents data from a project at Bull HN Information system which replaced inefficient C code for a control microprocessor. After system was completed, code inspection effectiveness was around 70%.

Grady and van Slack [Grady, 1994], report on experience from use of inspection at Hewlett Packard. In one of the company's division, code inspection typically found 60 to 70% of the defects.

Barnard and Price [Barnard, 1993] cite several references and report defect detection effectiveness for code inspection varying from 30 to 75%.

Glib and Graham [Glib, 1993] included experience data from various sources in their discussion on the benefit and cost of inspection. IBM Rochester Labs published [Glib, 1993] value of 60% for source code inspections.

Grady [Grady, 1994], performed cost and benefit analysis for different techniques, among them design and code inspections. He stated, that the average percentage of defects found during design inspection is 55%, and 60% for code inspection.

4.6.1.1 Effectiveness of Requirement Inspection

Glib [Glib, 1993] gives an estimate of defect removal effectiveness at requirement stage in the range of 50 to 70%. Since no further data are available for the requirement inspection, we have taken as an intuitive estimate of max value, most likely value and minimum value as: 70%, 60%, and 50%.

4.6.1.2 Effectiveness of Design Inspections

The maximum value for the effectiveness of the design inspections is reported in [Glib, 1993] from IBM Rochester Labs. This was 0.84 (average for two type of design document). The minimum value reported by Jones [Jones, 1996] is 0.25 for informal design review. The most likely value is the mid point of the data industry mean reported in [Jones, 1996]. The final set of value is summarized in Table 4.1

4.6.1.1.3 Effectiveness of Code Inspections

For the minimum value the data value of 0.19 as reported by Franz [Franz, 1994] is used. The maximum value of 0.7 reported in [Weller, 1993] is used. For the most likely value, the data from [Grady, 1994] [Glib, 1993] [Jones, 1996] [Meyer, 1978] was used to produce an average. The final set of value is summarized in Table 4.1

Table 4.2, Defect Injection Rates

Phase	Defect Injection Rate
REQUIREMENT	1.2 PER PAGE
DESIGN	1.5 PER PAGE
CODING	4.7 PER *KNCSL

[Weider, 1998] (*KNCSL, Thousand noncommentary source line)

Jalote [Jalote, 2000] in his book "CMM in Practice" specifies a process capability base line for the process executing software projects at Infosys. He specifies the value base of his experience during his tenure in the organization. Table 4.3 specifies these values.

Table 4.3, Defect Injection Rates (over all)

Phase	Defect Injection Rate
REQUIREMENT	0.6 –1.8 PER PAGE
Low Level Design	0.7 – 2.2 PER PAGE
Coding	0.02 – 0.12 Per Line of Code

[Jalote, 2000]

T able 4.1 Distributions of effectiveness

Defect detection Technique	Minimum value	Most likely value	Maximum value
Requirement inspection	0.50	0.60	0.70
Design inspection	0.25	0.57	0.84
Code inspection	0.19	0.57	0.70

The distribution of effectiveness in terms of its minimum, most likely, and maximum value give us the ability to specify in our model the effectiveness of inspector working on each individual phase as per their experience levels.

4.6.2 Defect Injection Rate

Weider D. Yu [Weider, 1998] reported a base value of injected fault density from his study on several program developed in 'C' language for the 5ESS switch software at Lucent Technologies. Table 4.2 specifies these base lines values.

4.6.3 Defect Amplification Factor

Glib and Graham [Glib, 1993] have estimated relative cost of fixing defect in subsequent phases by taking cost of fixing the same defect in the phase in which it is occurred. These cost factors are summarized in Table 4.4

Table 4.4 Cost factors for Defect Detection in subsequent Phases

	Requirement Analysis	High Level Design	Low Level Design	Coding	System Integration Stage	Operation /Usage
Requirement Analysis	**1.0**	**1.8**	**2.4**	**5.1**	**19**	**72**
High Level Design		**1.0**	**1.3**	**2.8**	**11**	**39**
Low Level Design			**1.0**	**2.2**	**8.0**	**30**
Coding				**1.0**	**3.8**	**14**
System Integration Stage					**1.0**	**3.7**
Operation /Usage						**1.0**

[Glib, 1993]

4.6.4 Chances of Detecting the Defect in the next Stage

Weider. D Yu. [Weider, 1997] reported that 84 % of the total fault developed during the Low Level Design Inspection on average could be detected in that phase, 4% in the next phase, that is in the code inspection, 4% in unit testing phase, and near about 6% in system testing phase.

We do not have any literature data to quantify how much of the requirement defect get detected in that phase it self, and how much in other phases. Information collected during informal discussion with inspectors working in SEPG Department of an established organization indicated that, about 70% of the requirement fault gets detected in its own stage, near about 10% at design stage, near about 8% at coding stage and rest in the subsequent stages.

CHAPTER FIVE

Model Description and User Interface

In this chapter we describe model for the simulation and user Interface for operating the game.

5.1 Model Boundary

5.1.1 Assumptions of Model

Model is based on the following assumptions:

- Time allocated to software inspection takes, time away from software development;
- Defects are generated uniformly throughout the document;
- Software inspection finds a high percentage of errors in the development life cycle;
- Error are detected only at the individual checking stage, that is at the preparation stage; and
- No errors are generated during rework.

5.1 .2 Limitations of Model

Model does not incorporate the following:

- The error injection rate of software developers decreases due to an increase in product knowledge acquired through the software inspection process;
- Software inspection lead to increased visibility of the amount of work completed;
- The time spent on inspection meeting (model does not reveal that inspection meeting time);

- Software developers achieve improved size estimates due to attention paid to size estimates during inspection; and
- Choice of defect detection method, that is the reading technique dimension of the software inspection process.

5.2 Decision Variables

The model allow direct manipulation of the following variables connected with the software inspection process:

- The time spent on each inspection task per unit of work to be inspected i.e. checking rate;
- The staffing level of experienced and inexperienced inspectors' for each phase to be determined for each time period (one week) in advance of that phase. The planned staffing level will remain constant for one time period (one week) and can be changed only in the next time interval;
- The percentage of task allocation; and
- The percentage of task that is to be checked by the super inspector.

5.3 Evaluation Variable

- Cost of the inspection process
- Total cost of the inspector time during inspection process
- Quality level achieved.

5.4 Rules of the Game:

- Checking rate can not exceed the specified maximum value;
- Volume of work to be allocated must be proportional to the schedule of the inspection process;
- Checking rate at any stage can not be zero; and

- For partially available manpower, attention is to be paid in allocating the percentage of the task.
- Manpower will be allocated to the inspection activity with the goal of meeting a specified turn-around time for evaluation of work products, at the desired evaluation rate. There will be, however, an upper limit to the percentage of total manpower that will be allocated to the evaluation activity. Bouergeosis [Bouergeosis, 1996] specifies an optimum team size of 3 to 5 inspector. That is 5 is the upper limit to the team size.
- The players are required to note down the values of decision variables that they selected for each predefined interval i.e. a week.

5.5 User - Interface of the game

The user can access the information about the product, process, and resources factor from the file **'Information.txt'**. The files contain these information's in the form of the Tables with different names. Table 5.1 named 'Product Factor' contains the information related to the product. Table 5.2 named 'Process Factor' contains the information related to process. Table 5.3 named 'Resources Factor' contains the information related to resources used. This information is compiled from Jalote [Jalote, 2000].

5.5.1 Interface with Input/Output file

5.5.1.1 Input file

The 'tool' has interface with two input files, named **insp.txt** and **proj.txt**. The input file **insp.txt** contains the information's about the availability of manpower. The data structure of file insp.txt is shown in figure 5.1. The second input file named **proj.txt** contains the information about the schedule of the inspection process. The data structure of file **proj.txt** is shown in figure 5.2.

Table 5.1 Product factor

Factor	Value

Size

Functional requirements	Small Medium Complex
Source code Size of design document Size of SRS	Small Medium Large
Process interaction	Many Few
Number of modules	Known Unknown

Criticality

Safety	Software is safety critical Hazards can be easily identified	Yes / No Yes / No
Mile stone	Objective	Yes/No

Off the shelf

Part of software acquired	Yes / No

Table 5.2 Process factors

Factor		**Value**

Development

Requirements	Ambiguous Expected to change	Yes / No Yes / No
Design	More design alternatives to select from	Yes / No

Operational

Interfaces	External & environmental	Yes / No
Process capability base line	Exists Can be determined	Yes / No Yes / No
Failures	Equally severe Design faults must be tolerated	Yes / No Yes / No

Table 5.3 Resource factor

Factor	Value

Personnel

Developers	Developers' experience with formal methods	Low (< 1year) Medium (1-2 years) High (> 2 years)
Inspectors	Skills, experience with formal method.	Low (< 1year) Medium (1-2 years) High (> 2 years)

Facilities

Support systems	Prototyping tools available	Yes / No

Insp_id	Exp_level	Avail_from	Avail_to
100	l	DD MM YYYY	DD MM YYYY
200	m	DD MM YYYY	DD MM YYYY
300	h	DD MM YYYY	DD MM YYYY
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.

COMMENTS:

- The first field have the value of inspector id
- The second field specify experience level of inspectors with 'l' specify inspector of low level experience, 'm' specify the inspector of moderate level experience, and 'h' with inspector of high level experience.
- The third and fourth field specifies their availability period.

Figure 5.1 Data structure of input file 'insp.txt'

Phase	Start date	End date
Requirement	DD MM YYYY	DD MM YYYY
Design	DD MM YYYY	DD MM YYYY
Coding	DD MM YYYY	DD MM YYYY

COMMENTS:

- The phase field specify the phase of the inspection process;
- The start date specify the actual start date of inspection; and
- The end date specifies the actual end date of inspection.

Figure. 5.2 Data structure of input file 'proj.txt'

5.5.1.2 Interactive allocation

Once the user completes the required information in both the input file, the tool will flash out the screen with identifier "**Availability Constraint**". The screen has the format shown in the figure 5.3. Next the user will be asked to fill information on the form shown in figure 5.4. After this the user will be asked to allocate manpower for the predefined interval i.e. a week in the form shown in the figure 5.5. Figure 5.6 represents flow of steps of the game during its execution. Figure 5.7 is flow chart representation of the sequence of steps involved in the design of the 'tool'.

5.5.1.3 Output files

The 'tool' contains two-output file, **weekresult.txt** and the **phaseresult.txt** these file are generally to store the result for the evaluation purpose, and have the form shown in figure 5.8 and figure 5.9.

==================== =Availability Constraint =======================

ıase	Exp level	From	Up to	Partial/full	Days	hour	week	last week day
·quirement	l	DD MM YYYY	DD MM YYYY	p	d1	h1	w1	d1'
·quirement	m	DD MM YYYY	DD MM YYYY	p	d2	h2	w2	d2'
·quirement	h	DD MM YYYY	DD MM YYYY	f	d3	h3	w3	d3'
·sign	l	DD MM YYYY	DD MM YYYY	p	d4	h4	w4	d4'
·sign	m	DD MM YYYY	DD MM YYYY	f	d5	h5	w5	d5'
·sign	h	DD MM YYYY	DD MM YYYY	p	d6	h6	w6	d6'
ding	h	DD MM YYYY	DD MM YYYY	p	d7	h7	w7	d7'
ding	m	DD MM YYYY	DD MM YYYY	f	d8	h8	w8	d8'
ding	l	DD MM YYYY	DD MM YYYY	p	d9	h9	w9	d9'

MMENTS:

- The phase field refers to the phase of the inspection process.
- The experience level field specifies the experience level
- Field From and Up to specifies the period of their availability in that particular phase.
- Field partial/full specifies whether the manpower is available fully (means for the whole schedule of that particular phase) or partially which indicate that the manpower is available for some portion of the respective phase schedule.
- Field days, hour, week specifies availability in terms of number of days, number of weeks, and number of hour.
- Field last week day refer to the partially available person, by specifying the number of days to which the partial manpower is available in that week in which he is partially available. For example if the value of this field is 2 and week field indicate 3 than it means the particular man power is available in the third week only for 2 days.

Figure. 5.3 Screen, "Availability Constraint"

You Are Required to fill up Following Information's:

The number of pages of complex complexity level of the requirement document:

The number of pages of normal complexity level of the requirement document:

The number of pages of complex complexity level of the design document:

The number of pages of normal complexity level of the design document:

The number of line of code of complex complexity level:

The number of line of code of normal complexity level:

COMMENTS:

- Line of code means the number of noncommentary source line of code.
- The pages specifies the single page of noncommentary information's

Figure 5.4 Input query form

For the Phase

For week 1 Availability:

The number of low-level experience: n1 [] []...[]

The number of moderate level experience: n2 [] []...[]

The number of high level experience: n3 [] [][] []...[]

COMMENTS:

The n1, n2, n3, represent the number of manpower available in that particular week, the subscripts were to account the partially available manpower if the value in all the subscripts is 7 then it's means that the manpower is fully available if in any of the subscripts have value less than 7 than its means that particular man power is partially available.

Figure 5.5-Week availability constraint screen

====== Requirement Phase ====== For week 1 You are Required to Fill up following Information: Number of low level experienced inspector: m1 (say) For first person please fill the following information: Number of pages of complex complexity level assign: Number of pages of normal complexity level assign: For person 1, specify the checking rate [Optimum: A-B, Max: C] . Up to m1th person Number of moderate level experienced inspector: m2 (say) Number of pages of complex complexity level left: Number of pages of normal complexity level left:	. Up to m2th person Number of high level experienced inspector: m3 (say) . . Up to m3th person End of week Super user intervention You are Required to Fill up following Information: The fraction of pages to be checked by the super user for the number of pages checked by utilized manpower Check rate for the super user: For week2 . . Up to The last week of the Phase

Continued

Phase Result:
Number of defects detected:
Number of defects escaped:
Cost of the inspection process:
Rework Hour:
Efficiency of the inspection process:

= = Design Phase = =

Phase Result:

= = Coding Phase = =

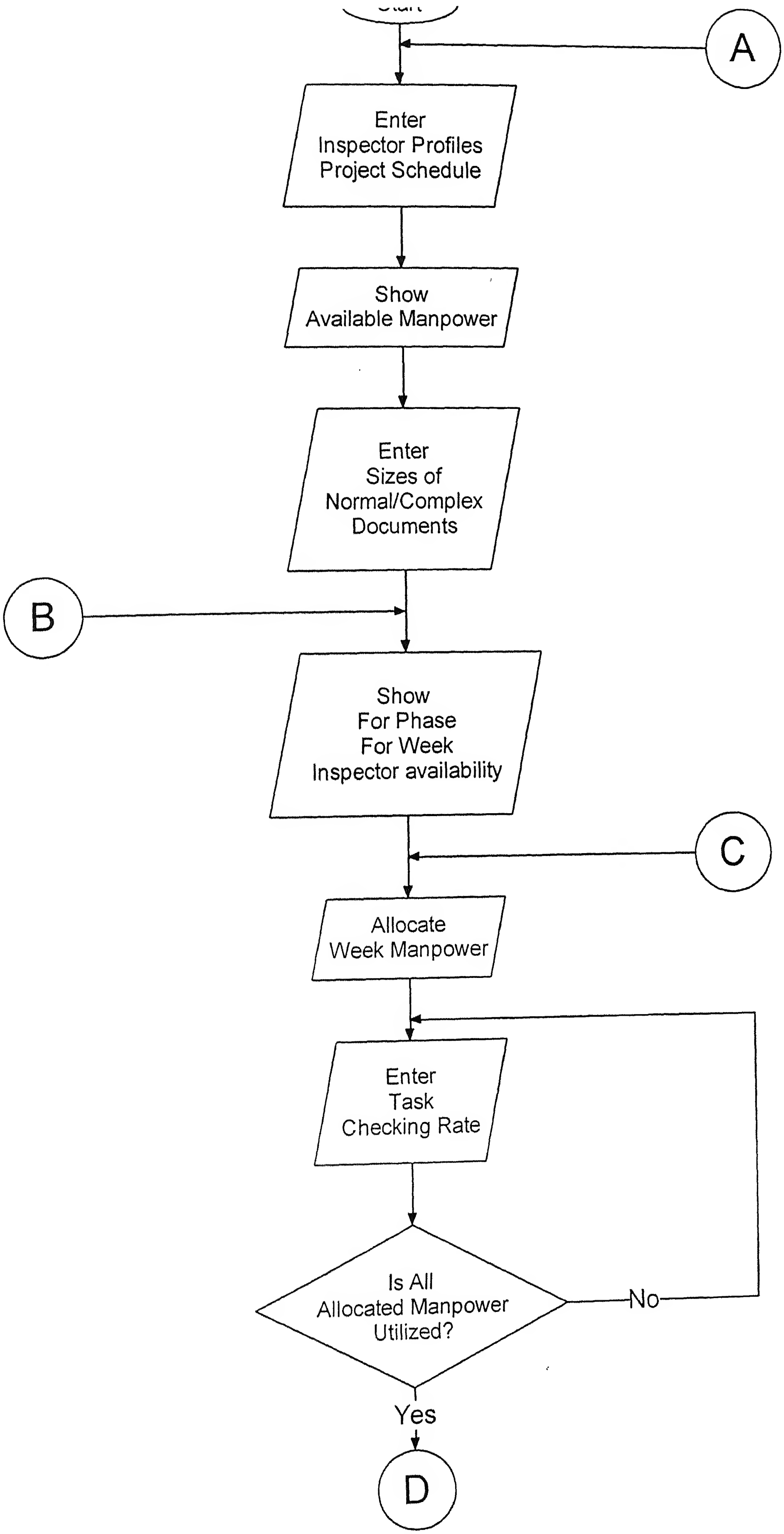
Phase Result:

= = End of the all three Phases
Net Results
Net cost of the inspection process
Net value of defect detected:
Net value of defect escaped:
----Please specify-----
You want to play more Y/N:

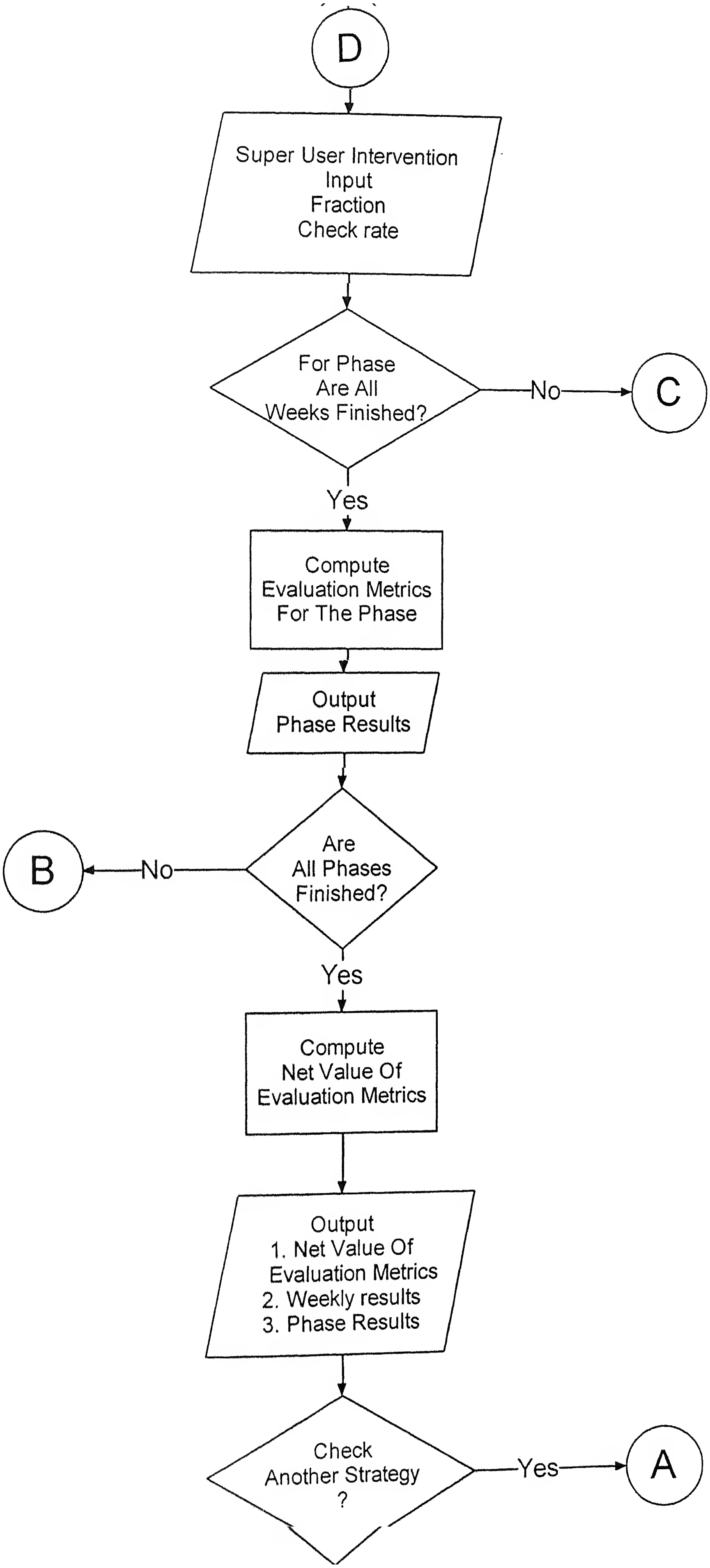
COMMENTS:

- After the completion of one predefined time period i.e. a week a provision of evaluation is provided, a super inspector inspect the fraction of pages/line of code that has been checked by the man power allocated on that particular week. That has some impact on cost as cost of super inspector gets included but the evaluation helps to maintain the effectiveness of the inspection process. However this alternative is optional.
- The Phase results will help the user to formulate strategy for the next phase.
- At the completion of all the three phase i.e. requirement, design and code the tool will flash out the net results on the screen.
- The last query give user the facility to go for one another strategy and hence facilitate users to learn from their experiences.

Figure 5.6 Execution of game



Start
A
Enter
Inspector Profiles
Project Schedule
Show
Available Manpower
Enter
Sizes of
Normal/Complex
Documents
B
Show
For Phase
For Week
Inspector availability
C
Allocate
Week Manpower
Enter
Task
Checking Rate
Is All
Allocated Manpower
Utilized?
No
Yes
D


D
Super User Intervention
Input
Fraction
Check rate
For Phase
Are All
Weeks Finished?
No
C
Yes
Compute
Evaluation Metrics
For The Phase
Output
Phase Results
Are
All Phases
Finished?
No
B
Yes
Compute
Net Value Of
Evaluation Metrics
Output
1. Net Value Of
Evaluation Metrics
2. Weekly results
3. Phase Results
Check
Another Strategy
?
Yes
A

weekresult.txt
The impact of week to week decisions for the phase are: Number of defects detected: Number of defects escaped: Cost of the inspection process: Rework Hour: Efficiency of the inspection process
COMMENTS: Week to week decision are provided to give user flexibility to check the impact of week-to-week decision.

Figure 5.8 format of output file 'weekresult.txt'

phaseresult.txt
The impact of phase to phase decision for phase are: Number of defects detected: Number of defects escaped: Cost of the inspection process: Rework Hour: Efficiency of the inspection process
COMMENTS: The phase results help users to plan his strategy for the next phase.

Figure 5.9 format for output file 'phaseresult.txt'

CHAPTER SIX

Conclusion

The objective of this research was to build a learning tool, which helps the software manager to learn to manage the software inspection process effectively.

The approach to developing this tool has been to build a model for the software inspection process by using the discrete defect removal model and incorporating in it, the features of the system dynamics modeling. The concept of system dynamics modeling allows variation to the input parameter of the model with respect to the system dynamics variables i.e. the experience level of inspectors, team size etc.

The specification of the software inspection process model is the accumulation of knowledge gathered from the literature and the industry regarding the cause-effect relationships that exist in the model and impact on cost and quality level achieved.

The model also accounts for the chances of defect detection of those defects, which escaped from previous phase, in to the next phase. Hence also considers the dependency of defect generated in the next phase with respect to inspection process effectiveness of the previous phase i.e. with the defect detected and the defect escaped in the previous phase.

The model is general enough to incorporate any development process i.e. the model parameter are flexible enough to interface the model with any kind of development process model.

As a simulator, the model allows a variety of management decision to be made by the users in response to impact of decision that users has taken in the previous phases i.e. it provides feedback to the users interactively to help him in taking better decisions.

These contributions provide the foundation for further study of inspection process improvement and their impact on the cost and quality level achieved.

Direction for the Future Research

- In the current research we have only considered the preparation stage of the defect detection stage i.e. defect is detected only at the individual preparation stage. However the meeting stage of the inspection process is vital one and quite useful for to detect those latent defect that has not been found at preparation stage, because of the synergic effect of group activity.

So the inclusion of this stage is an important proposition for the future work.

The data taken from the literature have following sources of variation:

- The data from the literature comes from different application contexts (e.g., different application domain) and it may have different level of precision due to different organization exhibiting different amount of rigor in their data collection.
- The unit of analysis of calculation reported in the literature also can vary. For example, in some reports, defect detection effectiveness is calculated across all work products for projects. In others, the effectiveness for each work product is calculated, and then the overall effectiveness is computed as the average over all effectiveness values.

It is important to capture this variability because it reflects the uncertainty we have in the values that have been obtained from the literature.

The future work can account these types of uncertainty by modeling each of the variables as distribution rather than a single value.

- In the current research we have not accounted for the difference in quality level of work product, which go through the defect detection phases. However in actual practice it is an important parameter.

Extending the model for the development process model and accounting the variable connected to the development process model can overcome this limitation.

REFERENCESES

1. [Abdel, 1989] Tarek K. Abdel-Hamid and Stuart E. Madnick, "Lessons Learned from Modeling the Dynamics of Software Development," *Communications of the ACM*, Vol. 32, No. 12, pp 45-55, 1989.
2. [Ackerman, 1989] Ackerman, A. F., Buchwald, L. S., and Lewsky, F. H. "Software Inspections: An Effective Verification Process," *IEEE Software*, 6(3): pp 31--36, 1989.
3. [Anita, 1994] Anita D. C, Danish J. Paulish. "Case Studies of Software Process-Improvement Measurement," Computer, pp 50-57, September 1994.
4. [Barbara, 1995] Barbara Kitchenham, Lesley Pi ckard, and Shari Lawrence Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software*, pp. 52-62, July 1995.
5. [Barnard, 1994] Barnard, J. and Price, A. "Managing Code Inspection Information," *IEEE Software*, 11 (2): pp 59–69, 1994.
6. [Bate, 1995] Roger Bate, Dorothy Kuhn, Curt Wells, et al, "A Systems Engineering Capability Maturity Model, Version 1.1," Software Engineering Institute, CMU/SEI-95-MM-003, htpp://www.sei.cmu.edu/SEMA/TR-95, November 1995.
7. [Basili, 1996] Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., and Zelkowitz, M. "The Empirical Investigation of Perspective-based Reading," Journal *of Empirical Software Engineering*, 2(1): 133–164, 1996.
8. [Basili, 1991] V. R. Basili and J. D. Musa, "The Future Engineering of Software: A Management Perspective," Computer 24, No. 9, 90-96, 1991.
9. [Basili, 1991] V. R. Basili and Barry T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, Vol. 27, January 1984.
10. [Belli, 1996] Belli, F. and Crisan, R. "Towards Automation of Checklist-based Code-Reviews," In *Proceedings f the International Symposium on Software Reliability Engineering* (Ref. Home page of Victor. R. Basili), 1996.

11. [Beizer, 1983] Beizer, Boris, *"Software Testing Techniques,"* Van Nostrand Reinhold Company New York, 1983.

12. [Bisant, 1989] Bisant, D. B. and Lyle, J. R. "A Two-Person Inspection Method to Improve Programming Productivity," *IEEE Trans. on Software Engineering*, 15(10): pp 1294 1304, 1989.

13. [Blakely, 1991] Blakely, F. W. and Boles, M. E. "A Case Study of Code Inspections," *Hewlett-Packard Journal*, 42(4): pp 58–63, 1991.

14. [Boehm, 1981] Barry W. Boehm, *"Software Engineering Economics,"* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

15. [Bourgeois, 1996] Bourgeois, K. V. "Process Insights from a Large-Scale Software Inspections Data Analysis", *Cross Talk, The Journal of Defense Software Engineering*, pp 17 23, 1996.

16. [Chernak, 1996] Chernak, Y. "A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement," IEEE *Trans. on Software Engineering*, 22(12): pp 866 874, 1996.

17. [Chichakly, 1993] Karim J Chichakly, "The Bifocal Vantage Point: Managing Software Projects From a Systems Thinking Perspective," *American Programmer*, pp. 18-25, May 1993.

18. [Chilarecge, 1992] R. Chilarege, I. Bhandari, " Orthogonal Defect Classification-A Concept for in Process Measurement," IEEE Trans. On Software Engineering, Vol. 18, pp 943-956, Nov. 1992.

19. [Christenson, 1990] Christenson, D. A., Steel, H. T., and Lamperez, A. J. "Statistical quality control applied to code inspections," *IEEE Journal Selected Areas in Communication*, 8(2): pp 196–200, 1990.

20. [Copper, 1978] Copper, J. D. "Corporate level software Management," IEEE Trans. on Software Engineering, SE-4, No. 4 319-325, July 1978.

21. [Copper, 1993] Kenneth G. Cooper and Thomas W. Mullen, "Swords and Plowshares: The Rework Cycles of Defense and Commercial Software Development Projects," *American Programmer*, pp. 41-51, May 1993.

22. [Collofello, 1989] Collofello, J. S. and Woodfield, S. N. (1989). "Evaluating the effectiveness of reliability-assurance techniques," Journal *of Systems and Software*, 9: pp 191–195, 1989.

23. [Cox, 1990] B. J. Cox, "Planning the Software Industrial Revolution," *IEEE Software*, pp. 25-42, November 1990.

24. [Curtis, 1988] B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, Vol. 31, No. 11, pp. 1268-1287, 1988.

25. [Doolan, 1992] Doolan, E. P. "Experience with Fagan's Inspection Method. *Software–Practice and Experience,"* 22(3): pp 173–182, 1992.

26. [Dyer, 1992a] Dyer, M. "*The Cleanroom Approach to Quality Software Development,*" John Wiley and Sons, Inc 1992.

27. [Dyer, 1992b] Dyer, M. Verification-based Inspection. In Proceedings of the 26th Annual Hawaii International Conference on System Sciences, htpp://www.davidfrico.com, 1992.

28. [Fagan, 1976] Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development," IBM *Systems Journal*, 15(3): pp 182–211, 1976.

29. [Fagan, 1986] Fagan, M. E. "Advances in Software Inspections," *IEEE Trans. on Software Engineering*, 12(7): pp 744–751, 1986.

30. [Fowler, 1986] Fowler, P. J. In-process Inspections of Work products at AT&T. *AT&T Technical Journal*, 65(2): 102–112, 1986.

31. [Franz, 1994] Franz, L. A. and Shih, J. C. "Estimating the Value of Inspections and Early Testing for Software Projects," CS-TR- 6, Hewlett–Packard Journal, pp 39-43, 1994.

32. [Fraunhofer, 1998] Fraunhofer Institute for Experimental Software Engineering."Aninspectionbibliography,"1998. http://www.iese.fhg.de/ISE/Inspbib/inspection.html

33. [Gehring, 1977] P. F. Gehring, Jr. and V. W. Pooch, "Software Development Management," *Data Management*, pp. 14-38, February 1977.

34. [Gilb, 1993] Gilb, T. and Graham, D. "*Software Inspection*", Addison-Wesley, New York, 1993.

35. [Gintell, 1995] Gintell, J., Houde. M., and McKenney, R. "Lessons learned by building and using Scrutiny, a collaborative software inspection system," In *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering*,. www.davidfrico.com , 1995

36. [Grady, 1994a] Grady, R. B., "Successfully applying software metrics," *IEEE Computer*, 27(9): pp 18–25, 1994.

37. [Grady, 1994b] Grady, R. B. and van Slack, T., "Key Lessons in Achieving Widespread Inspection Use," IEEE *Software*, 11(4): pp 46–57, 1994.

38. [Goel, 1985], " A. L. Goel "Software reliability Models: Assumptions, Limitations, and Applicability," IEEE Trans. of Software Engineering, SE-11, No. 12, pp 319-325, 1985.

39. [Gordon, 1978]. Gordon, Geoffrey, *"System Simulation"*, Prentice-Hall, Inc., 1978.

40. [Howard, 1995] Howard A. Rubin, Margaret Johnson, and Ed Yourdon, "Software Process Flight Simulation Dynamic Modeling Tools and Metrics," *Information Systems Management*, summer 1995.

41. [Howard, 1994] Howard A. Rubin, Margaret Johnson, and Ed Yourdon, "With the SEI as My Copilot Using Software Process 'Flight Simulation' to Predict the Impact of Improvements in Process Maturity," *American Programmer*, pp. 50-57 September 1994.

42. [Humphrey, 1995] Humphrey, W. H.). *"A Discipline for Software Engineering,"* Addison-Wesley, New York 1995.

43. [Iona, 1996], Iona, Rus "Modelling the Impact on Project Cost and Schedule of Software Engineering Practices for Achieving and Assessing Software Quality Factors," a Ph.D. Dissertation Proposal, Arizona State University Department Of Computer Science and Engineering, htpp://www.asu.edu/sdm, 1996.

44. [Jalote, 2000] P. Jalote "CMM in Practice: Process for Executing Software Projects at Infosys," Addison-Wesley, New York, 2000.

45. [Johnson, 1997] Johnson, P. M. and Tjahjono, D., "Assessing Software Review Meetings: A Controlled Experimental Study Using CSRS," In *ACM Press*, http://www.iese.fhg.de/ISE/Inspbib, 1997

46. [Jones, 1994] Jones, C. "Gaps in the object-oriented paradigm," *IEEE Computer*, 27(6): pp 90–91, 1994.

47. [Jones, 1996] Jones, C., "Software Defect-Removal Efficiency," IEEE *Computer*, 29 (4): pp 94 95 1996.

48. [Kan, 1991] S. H. Kan, "Modelling and Software Development Quality," in IBM *Systems Journal*, 30 (3): pp 351-361, 1991

49. [Kelly, 1992] Kelly, J. C. Sherif, J. S., and Hops, J., "An Analysis of Defect Densities found during Software Inspections," *Journal of Systems and Software*, 17: 111–117, 1991.

50. [Kellner, 1991] Marc I. Kellner, "Software Process Modeling Support for Management Planning and Control," *Proceedings of the First International Conference on the Software Process*, pp. 8-28, 1991.

51. [Kibbee, 1964] Jome M. Kibbee, C. J. Craff "*Management Games: A new Technique for Executive Development*," Chapman &Hall, Ltd., London, 1962.

52. [Knight, 1991] Knight, J. C. and Myers, E. A. "Phased Inspections and their Implementation", *ACM SIGSOFT Software Engineering Notes*, 16(3): pp 29–35, 1991.

53. [Knight, 1993] Knight, J. C. and Myers, E. A. "An Improved Inspection Technique," *Communications of the ACM*, 36(11): pp 51–61, 1993.

54. [Laprie 1992] Laprie, J.C., ed. Dependable Computing and Fault Tolerant Systems. Vol. 5, "Dependability: Basic Concepts and Terminology in English, French, German, Italian, and Japanese" Springer-Verlag, New York, 1992.

55. [Lamport 85] Lamport, L. and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," Journal of the ACM, vol. 32 no. 1, 52-78, January 1985.

56. [Littlewood, 1992] B. Littlewood and L. Strigini, "The Risk of Software," Scientific American, pp 62-75, November 1992.

57. [Lin, 1993] Chi Y. Lin, "Walking on Battlefields: Tools for Strategic Software Management," *American Programmer*, pp. 33-40, May 1993.

58. [Linger, 1979] Linger, R. C., Mills, H. D., and Witt, B. I. (1979). "*Structured Programming: Theory and Practice*," Addison-Wesley New York, 1979.

59. [Kellener, 1992] M. I. Kellener and J. W. Over, "A Software Quality Improvement Frame Work," www.sei.cmu.edu/SEMA/TR-92, 1992

60. [Kim, 1995] Kim, L. P. W., Sauer, C., and Jeffery, R. A Framework for Software Development Technical Reviews. *Software Quality and Productivity: Theory, Practice, Education and Training,* 1995.

61. [Macdonald, 1995] Macdonald, F. and Miller, J. "Modeling Software Inspection Methods for the Application of Tool Support". Technical Report RR-95-196 [EFoCS-16-95], Empirical Foundations of Computer Science (EFoCS), University of Strathclyde, UK, 1995.

62. [Martin, 1990] Martin, J. and W.T.Tsai. "N-fold Inspection: A Requirements Analysis Technique," *Communications of the ACM*, 33(2): 225–232, 1990.

63. [Mashayekhi, 1993] Mashayekhi, V., Drake, J. M., Tsai, W. T., and Riedl, J. "Distributed, Collaborative Software Inspection," IEEE *Software*, 10:66–75, 1993.

64. [McCabe, 1976] McCabe, T. J. "A Complexity Measure," IEEE *Trans. on Software Engineering*, 2(4): 308–320, 1976.

65. [McGibbon, 1996] McGibbon, T. "A Business Case for Software Process Improvement," Technical Report, F30602-92-C-0158, Data & Analysis Center for Software (DACS). URL: http://www.dacs.com/techs/roisoar/soar.html, 1996.

66. [Misra, 1983] P. N. Misra, "Software Reliability Analysis," IBM Systems Journals 22 No. 3, pp 262-270, 1983.

67. [Myers, 1978] Myers, G. J., "A Controlled Experiment in Program Testing and Code Walkthroughs / Inspections," *Communications of the ACM*, 21(9): 760–768 1978.

68. [NASA, 1993] National Aeronautics and Space Administration "Software Formal Inspection Guidebook," Technical Report, NASA-GB-A302, National Aeronautics and Space Administration. http://satc.gsfc.nasa.gov/fi/fipage.html, 1993.

69. [Ohba, 1984] M. Ohba, "Software Reliability Analysis Models," IBM Journal of Research of Development 28, No.4, pp 428-443, 1984.

70. [Oliver, 1998] Oliver, Jean –Marc, "An Encompassing Life-Cycle Centric Survey of Software Inspection" ISERN-98-32, htpp://www.iese.fhg.de, 1998

71. [Parnas, 1987] Parnas, D. L. (1987). "Active Design Reviews: Principles and Practice," *Journal of Systems and Software*, 7: 259–265, 1987.

72. [Paulk, 1993] Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software*, pp. 18-27, July 1983.

73. [Porter, 1997a] Porter, A. A. and Johnson, P. M. "Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies," *IEEE Trans. on Software Engineering*, 23(3): pp 129-144, 1997.

74. [Porter, 1998] Porter, A. A., Siy, H., Mockus, A., and Votta, L. "Understanding the Sources of Variation in Software Inspections," *ACM Transaction on Software Engineering and Methodology*, 7(1): pp 49-79, 1998

75. [Porter, 1997b] Porter, A. A., Siy, H. P., Toman, C. A., and Votta, L. G. "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," IEEE *Trans. on Software Engineering*, 23(6): 329–346, 1997.

76. [Porter, 1997c] Porter, A. A. and Votta, L. G. "What Makes Inspections Work?," *IEEE Software*, pp 99–102, 1997.

77. [Porter, 1995] Porter, A. A., Votta, L. G., and Basili, V. R. "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," IEEE *Trans. on Software Engineering,* 21(6): 563–575, 1995.

78. [Pressman, 1982] R. S. Pressman, " *Software Engineering: A Practitioner's Approach,*" Mc Graw Hill Singapore, 1982.

79. [Putnam, 1978] Lawrence H. Putnam, "General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Trans. on Software Engineering*, Vol. SE-4, No. 4, pp. 345-361, July 1978.

80. [Richardson, 1982] G. P. Richardson, "*Sources of Rising Product Development Times*," Technical Report D-3321-1, SD Group, Cambridge, MA, M.I.T., 1982.

81. [Roberts, 1964] Edward B. Roberts, "*The Dynamics of Research and Development*," Published Ph.D. dissertation, M.I.T., Cambridge, MA, 1964

82. [Roberts, 1981] Edward B. Roberts, "A Simple Model of R & D Project Dynamics," *Managerial Applications of System Dynamics*," Edited by Edward B. Roberts, The M.I.T. Press, Cambridge, MA, 1981.

83. [Rombach, 1990] H. Dieter Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990.

84. [Russell, 1991] Russell, G. W. "Experience with Inspection in Ultralarge Scale Developments," IEEE *Software*, 8(1): 25–31, 1991.

85. [Stark, 1994] George Stark, Robert C. Durst, and C. W. Vowell, "Using Metrics in Management Decision Making," *Computer*, pp. 42-48, September 1994.

86. [Svendsen, 1992] Svendsen, F. N. "Experience with inspection in the maintenance of software," In *Proceedings of the 2nd European Conference on Software Quality Assurance,* htpp:// www.iese.fhg.de, 1992.

87. [Tervonen, 1996] Tervonen, I. "Support for Quality-Based Design and Inspection," IEEE *Software*, 13(1): 44–54, 1996.

88. [Tjahjono, 1996] Tjahjono, D. "Exploring the Effectiveness of Formal Technical Review factor with CSRS, Collaborative Software Review System," PhD thesis, Department of Information and Computer Science, University of Hawaii htpp:// www.davidfrico.com, 1996.

89. [Wheeler, 1997] Wheeler, D. A., Brykczynski, B., and Jr., R. N. M. Software Peer Reviews. In Thayer, R. H., editor, Software Engineering Project Management. IEEE Computer Society, 1997.

90. [Weider, 1997] Weider D. Yu, Alvin Barshefsky, and Steel T. Huang, "An Empirical Study of Software Fault at a Personal Level in a Very Large Software Development Environment," Bell Labs Technical Journal, pp 221-231, Summer 1997

91. [Weider, 1998] Weider D. Yu, "A Software Fault Prevention Approach in Coding and Root Cause Analysis," Bell Labs Technical Journal, pp 3-21 April-June 1998.

92. [Weinberg, 1984] Weinberg, G. M. and Freedman, D. P. "Reviews, Walkthroughs, and Inspections," IEEE *Trans. on Software Engineering*, 12(1): 68–72, 1984.

93. [Weller, 1993] Weller, E. F. "Lessons from Three Years of Inspection Data," IEEE *Software*, 10(5): 38–45, 1993.

94. [Yourdon, 1989] Yourdon, E. "*Structured Walkthroughs*," Prentice Hall, New York, 1989

95. [Zmud, 1980] Zmud, R. W. " Management of Large Software Development Efforts," MIS Q4, 2 June, 1980

Appendix -A

Definitions of following key termed used are based on those in Technical Report CMU/SEI-95-TR-021ESC-TR-95-021 December 1995 [www.sei.cmu.edu]

A.1 Fault:

A fault is an adjudged or hypothesized of cause an error. As shown in figure A.1, they can be classified along five main axes: phenomenological causes nature, phase of creation, system boundary and persistence.

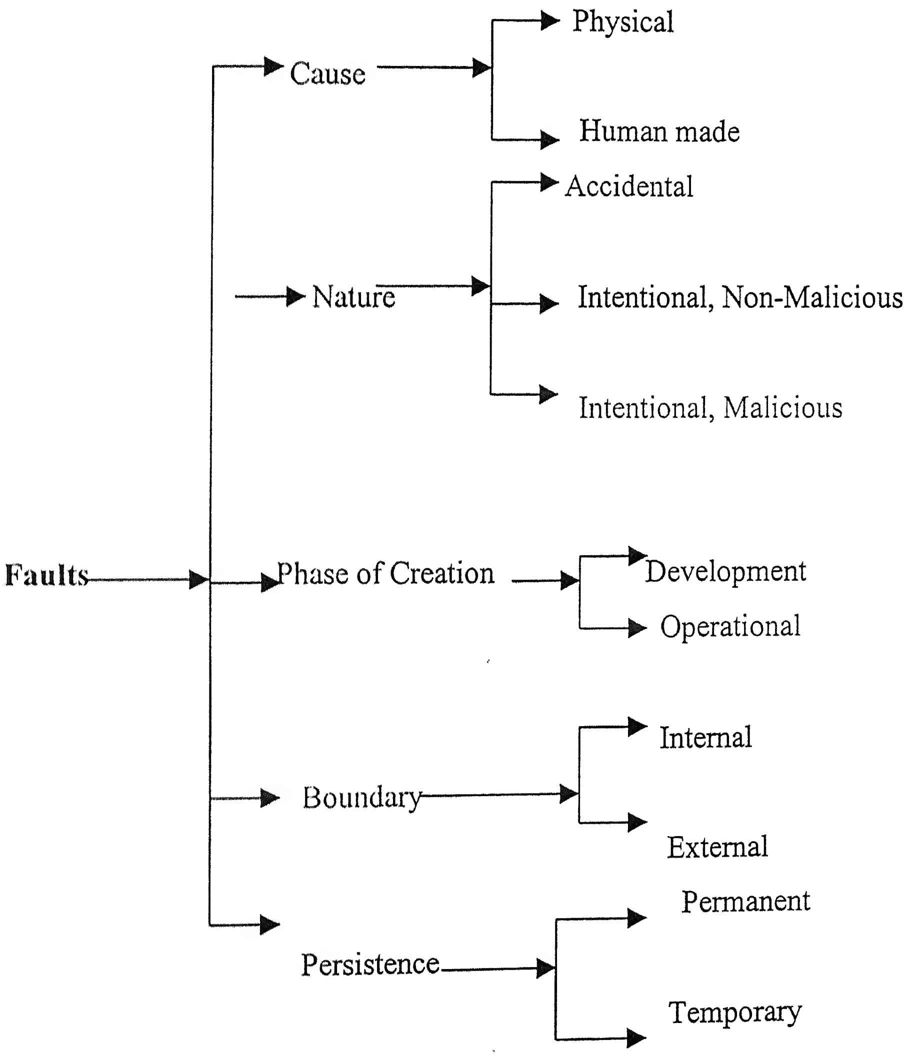


Figure. A.1 Fault Classes [Laprie, 1992]

Physical fault are those fault that occur, because of adverse physical phenomena (e.g., lighting). Human-made faults result from human imperfection and may be the result of many factors, singly or in cooperation, including poor design, inadequate requirements, or misuse. Accidental fault appear to be or created by chance. Intentional faults are created deliberately, with or without malicious intent.

Faults can be created at development time, or while the system is running (field fault) Faults can be internal faults, which are those parts of the internal state of the system which, when invoked, will produce an error.

A.2 Error

An error is a system that is liable to lead to a failure if not corrected. Whether or not it will lead to a failure is a function of three major factors [Laprie 1992]:

1. The redundancy (either designed or intent)
2. The system activity (the error may go away before it causes damages)
3. What the user deems acceptable behavior. For instance, in data transmission there is no notion of acceptable error rate.

A.3 Failures

A system fails when its behavior differ from its intended behavior. A system can fail in many different ways. As shown in figure A.2, the so-called "failure modes" of a system may be loosely grouped into three categories; domain failure, perception by the users, and consequences of the environment.

Domain failure includes both value failure and timing failures. A value failure occurs when improper value is computed. One inconsistent with proper execution of the system. Timing failures occurs when the system delivers its service either too early or too late. An

extreme form of timing failure is the halting failure-the system no longer delivers any service to the user. It is difficult to distinguish very late timing failure from a halting failure.

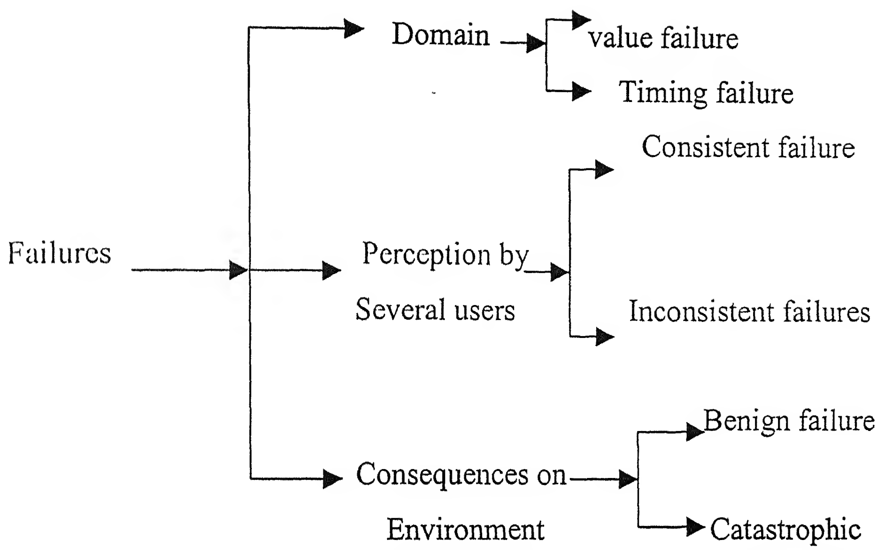


Figure. A.2 Failure Classification [Lompart, 1985]

There are two types of perception failures. A failure can be either consistent failures, or inconsistent. In the case of consistent failure, all system users have the same perception of a failure. In the case of inconsistent failure, the perception of systems user differs from each other [Lompart, 1985].

Finally, failure can be graded according to the consequences on the environment. Although extremely difficult to measure, failure can be classified as in the range **benign** to **catastrophic**. A system that can only fail in a benign manner is termed as fail-safe.

A.4 What is software fault?

Software is essentially an instrument for transforming a discrete set of inputs into a discrete set of outputs. It comprises of set of coded statement whose function is to evaluate an expression and store the result in a temporary or permanent location, decide which statement to execute next or to perform input/output operation [Chillarge, 1992].

Since largely, human being develop software, the finished product is often imperfect. It is imperfect in a sense that a discrepancy exists between what the software can do versus what the user or the computing environment wants it to do. The computing environment refers to, the physical machine, operating system, compiler and translator utilities, etc. These discrepancies are software faults. The programmer can attribute their faults to an ignorance of the user requirements, ignorance rules of computing environment, and to poor communication of software requirements between the user and the programmer or poor documentation of the software fault.

A.5 Classification of Defects

In the software development process, although, activities are broadly divided into design, code, and test, each organization have its own variations. It is also the case that process stages in several instance's may overlap while different releases may develop in parallel, further different process stages can be carried out by different people and some time by different organizations. Therefore, for classification to be widely applicable, the classification scheme must have consistency between the stages. Without consistency, it is almost impossible to look at trends across stages. Ideally, the classification should also be quite independent of the specifics of product and organization. If the classification is both consistent across phases and independent of the product, it tends to be fairly process invariant. One of the pitfalls of classifying defect is that, it is a human process, and is subject to the usual problem of human error, confusion, and general distaste if the use of the data is not well understood [Chillarege, 1992]. For example, a one-character error in a source statement changes the statement, but unfortunately, it passes syntax testing. In a result data are corrupted in an area far removed the actual fault. That in turn led to an improperly executed function. Is this a typing defect a coding defect or functional defect? The history of software engineering is populated with numerous example of using metrics to better manages and improve development process. The classification of defect to identify key component is also common and good exposition of it can be found in reference [Gardy, 1992]. There is also a proposed draft of an IEEE Standard on

classification. One such classification, developed by Boris Beizer is published in his book "Software Testing Technique" [Beizer, 1983].

A.5.1 Type Classification

The type of a defect is dependent on the phase in which it occurs. The following is a general classification of the defects during the various phases of the software development cycle.

A.5.1.1 Requirement Phase:

- **Incorrect Requirements**

This occurs when a requirement or a part of it is incorrect. This may be due to misunderstanding of the user expectation.

- **Undesirable Requirements**

The requirement stated is correct but is not desirable due to technical feasibility, design, implementation or cost consideration.

- **Requirements not needed.**

The user does not need the stated functionality or feature. Adding this requirement does not significantly increase the functionality of the product.

- **Inconsistent requirements.**

These requirements contradict some other requirement.

- **Ambiguous/Incomplete requirements.**

The requirement or part of it is ambiguous. It is not possible to implement the stated requirement

- **Unstable requirements.**

The requirement is likely to change during implementation

- **Standard violation**

The standard set by the organization or the client for analysis not followed.

A.6 Severity Classification

The severity classification measures the impact of the defect. During analysis, design and coding phases the defects might be classified as:

- Major defects, which have substantial, impact on several processes or subsystems. The correction activity involves changing the design of more than one process or subsystem.
- Minor defects that have impact on one process or subsystem. The correction activity will be local to that process or subsystem.
- Suggestion towards improvements

Appendix-B

Source Code

```
//include directives//

#include<stdio.h>
#include<time.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>

//Define Directives//
#define max_insp 15
#define max_phase 3
#define proj_file "proj.txt"
#define insp_file "insp.txt"
#define avail_file "avail.txt"
#define phase_result "phr.txt"
#define week_result "wer.txt"

  //Global structures//
struct date {
    int month;
    int day;
    int year;
};
// Data Structure of input file insp.txt//
struct {
    int id;
    char exp[2];
    struct date from;
    struct date to;
} insp[max_insp];
//Data Structure of input file proj.txt//
struct {
    char phase[8];
    struct date from;
    struct date to;
} proj[5];
//Data Structure of screen "Availability Constraint"//
struct {
    char phase[8];
    char exp[2];
    struct date from;
    struct date to;
```

```
    char f_p;
    int hours;
    int days;
    int weeks;
    int lwd;
} avail[max_insp];

struct l_h_m {
    int partial, full;
};

struct {
    struct l_h_m l,h,m;
} r_ment,design,code;

//Global Variables//
int i,j,k;
int countweeks[3];
int count[max_phase][3];
int liaf; // Variable representing last line in file avail.txt//
char *lmh[3]={"Low Level", "Moderate Level", "High Level"};

char *rdc[3]={"Reqment", "Design", "Coding"};
int persons[9];

// Hard coded values of defect injection rates of phase's
requirement, design and code//

float def_inj_rate[3]={1.2,1.5,0.0049};

float nc, A, B, C, D, E, F;
int max_weeks[3];
int so_n[3],so_c[3];
float su_cph=500;

// hard coded cost of super user in Rs. Per hour//
float su_p[3];
int su_cr[3];
float su_c[3], su_f[3];
float net_su_c;
int lmh_p[3];
int loop[max_phase];

int ****database;
```

```
//Global Function Declarations//

int is_leapyear(int);
int days_till_endyear(int , int , int );
const int days_in_month[12]= {31, 28, 31, 30,31,
30,31,31,30,31,30,31};
void search(void);
int finddays(int, int, int, int, int, int);
int comp(struct date,struct date);
void output(void);
void getavail(void);
void flash(void);
void builddatabase(void);
void asksizes(void);
void getuserdata(void);
void findmaxweeks(void);
void copyarray(float *, float *);


void main()
{
    FILE *fin1,*fin2;
    fin1=fopen(insp_file,"r");
    fin2=fopen(proj_file,"r");

    if(fin1==NULL) printf("error opening inspector file \n");
    if(fin2==NULL) printf("error opening project file \n");

    for(i=0;i<max_insp;i++)

((fscanf(fin1,"%d%s%d%d%d%d%d%d",&insp[i].id,&insp[i].exp,&insp[i]
.from.day,&insp[i].from.month,&insp[i].from.year,&insp[i].to.day,&
insp[i].to.month,&insp[i].to.year)));

    for(j=0;j<max_phase;j++)

((fscanf(fin2,"%s%d%d%d%d%d%d",proj[j].phase,&proj[j].from.day,&pr
oj[j].from.month,&proj[j].from.year,&proj[j].to.day,&proj[j].to.mo
nth,&proj[j].to.year))) ;

    fclose(fin1);
    fclose(fin2);

    search();
    output();
    flash();
```

```
// function to flash Availability Constraint on the screen//


    asksizes();
    findmaxweeks();
    builddatabase();
    getuserdata();

}

// end of main  //

// Function to compare given dates in file insp.txt, proj.txt//

int comp(struct date a,struct date b)

{
    if(a.year>b.year)
        return 1;
    else if(a.year <b.year)
        return -1;
    else if(a.year ==b.year)
    {
        if(a.month>b.month)
            return 1;
        else if (a.month <b.month)
            return -1;
        else if(a.month ==b.month)
        {
            if(a.day>b.day)
                return 1;
            else if (a.day <b.day)
                return -1;
            else if(a.day ==b.day)
                return -1;
        }
    }

}
// End of comparison //

// Function to check availability of manpower in given schedule of
project//

void search(void)

{
```

```
int l,m,p;
char phasename[8];

for(l=0;l<max_phase;l++)
      for(m=0;m<3;m++)
           count[l][m]=0;

k=0;
for(m=0;m<max_phase;m++)
{
    for(l=0;l<max_insp;l++)
    {
        if((comp(insp[l].from,proj[m].from)==-1)
            && ( comp(insp[l].to,proj[m].to)==1))
        {
            strcpy(avail[k].phase,proj[m].phase);
            strcpy(avail[k].exp,insp[l].exp);
            avail[k].from=proj[m].from;
            avail[k].to=proj[m].to;
            avail[k].f_p='f';
            ++k;
        }
        else if(((comp(insp[l].from,proj[m].from)==1)
            && ( comp(insp[l].to,proj[m].to)==1))
            && ( comp(insp[l].from,proj[m].to)==-1))
        {
            strcpy(avail[k].phase,proj[m].phase);
            strcpy(avail[k].exp,insp[l].exp);
            avail[k].from=insp[l].from;
            avail[k].to=proj[m].to;
            avail[k].f_p='p';
            ++k;
        }
        else if(((comp(insp[l].from,proj[m].from)==-1)
            && ( comp(insp[l].to,proj[m].to)==-1))
            && ( comp(insp[l].to,proj[m].from)==1))
        {
            strcpy(avail[k].phase,proj[m].phase);
            strcpy(avail[k].exp,insp[l].exp);
            avail[k].from=proj[m].from;
            avail[k].to=insp[l].to;
            avail[k].f_p='p';
            ++k;
        }
        else if((comp(insp[l].from,proj[m].from)==1)
            && ( comp(insp[l].to,proj[m].to)==-1))
        {
```

```
                    strcpy(avail[k].phase,proj[m].phase);
                    strcpy(avail[k].exp,insp[l].exp);
                    avail[k].from=insp[l].from;
                    avail[k].to=insp[l].to;
                    avail[k].f_p='p';
                    ++k;
                }
            }
        }

        liaf=k;

        for(p=0;p<liaf;p++)
        {

avail[p].days=finddays(avail[p].from.day,avail[p].from.month,
                avail[p].from.year,avail[p].to.day, avail[p].to.month,
                avail[p].to.year);

// Function to find Availability in terms of number of days//
        }
        for(p=0;p<liaf;p++)
        {
            if(avail[p].f_p=='f')
            {
                avail[p].lwd=7;
                if(avail[p].days%7==0)
avail[p].weeks=(int)(avail[p].days/7);
                else avail[p].weeks=(int)((avail[p].days/7)+1);
            }
            else
            {
                avail[p].weeks=(int)((avail[p].days/7)+1);
                avail[p].lwd=avail[p].days%7;
            }
        }

        for(p=0;p<liaf;p++)
        {
            strcpy(phasename,avail[p].phase);
            if(phasename[0]=='r')
            {
                if(avail[p].exp[0]=='l') count[0][0]++;
                if(avail[p].exp[0]=='m') count[0][1]++;
                if(avail[p].exp[0]=='h') count[0][2]++;
            }
            if(phasename[0]=='d')
```

```
        {
            if(avail[p].exp[0]=='l') count[1][0]++;
            if(avail[p].exp[0]=='m') count[1][1]++;
            if(avail[p].exp[0]=='h') count[1][2]++;
        }
        if(phasename[0]=='c')
        {
            if(avail[p].exp[0]=='l') count[2][0]++;
            if(avail[p].exp[0]=='m') count[2][1]++;
            if(avail[p].exp[0]=='h') count[2][2]++;
        }
    }
}

// Function to find number of days between two dates//


int finddays(int fd, int fm, int fy, int td, int tm, int ty)

{
    int first, j,t1, last;
    long middle;
    first=days_till_endyear(fd, fm, fy);
    t1=days_till_endyear(td, tm, ty);
    if( fy==ty)   return (first-t1);
    else
    {
        for (middle=0, j=fy+1; j<ty; j++)
            middle+=( is_leapyear (j)) ? 366 :365;
        last=365-t1;
        if (is_leapyear(ty))
            last++;
        return (first+middle+last);
    }
}

int is_leapyear(int y) // Function to check leap year condition//
{
    if ((y % 4)!=0) return 0;
    if((y % 100==0)&&(y %400 !=0)) return 0;
    return 1;
}
int days_till_endyear (int d, int m, int y)
{
    int total, j, days[12];
    for (j=0; j<=11;j++)
       days[j]=days_in_month[j];
```

```
        if(is_leapyear(y)) days[1]=29;
        for( total=days[m-1]-d+1, j=m; j<=11; j++)
         total+= days[j];
    return total;
}
```

// This write the details in the to an output file, avail.tx //

```
void output(void) {
    FILE *fout;
    int p;
    fout=fopen("avail.txt","w");

    for (p=0;p<k;p++)
        fprintf(fout,"%8s %2s   %2d %2d %2d      %2d %2d %2d
%c  %2d
%d\t%d\t%d\n",avail[p].phase,avail[p].exp,avail[p].from.day,avail[
p].from.month,avail[p].from.year,avail[p].to.day,avail[p].to.month
,avail[p].to.year,avail[p].f_p,avail[p].days,8*avail[p].days,avail
[p].weeks,avail[p].lwd);
}
void flash(void)
{
    int p;

    printf("Phase      Exp  From Date    To Date       f/p  Days
Hours\tWeeks\n");

printf("_______________________________________________________________
___\n");
    printf("\n");

    for (p=0;p<9;p++)
        printf("%8s %2s   %2d %2d %2d      %2d %2d %2d         %c   %2d
%d\t%d\t%d\n",

avail[p].phase,avail[p].exp,avail[p].from.day,avail[p].from.month,
        avail[p].from.year,avail[p].to.day,avail[p].to.month,

avail[p].to.year,avail[p].f_p,avail[p].days,8*avail[p].days,
        avail[p].weeks,avail[p].lwd);

}
void asksizes(void)
```

```
// Function to ask user to fill up input query form//

{
    printf("\nSize of the Normal Reqment document : ");
    scanf("%d",&so_n[0]);
    printf("\nSize of the Complex Reqment document: ");
    scanf("%d",&so_c[0]);
    printf("\nSize of the Normal Desgin  document : ");
    scanf("%d",&so_n[1]);
    printf("\nSize of the Complex Design document : ");
    scanf("%d",&so_c[1]);
    printf("\nLines   of  Normal code              : ");
    scanf("%d",&so_n[2]);
    printf("\nLines   of  Complex code             : ");
    scanf("%d",&so_c[2]);
}

// Function to find maximum number of week in schedules of each
phases//

void findmaxweeks(void)
{
    int p,temp;
    char phasename[8];
    for(p=0;p<9;p++)
    {
        strcpy(phasename,avail[p].phase);
        if(phasename[0]=='r') countweeks[0]++;
        if(phasename[0]=='d') countweeks[1]++;
        if(phasename[0]=='c') countweeks[2]++;
    }

    temp=avail[0].weeks;
    for(p=0;p<countweeks[0]-1;p++)
    {
        if(avail[p+1].weeks>=temp) temp=avail[p+1].weeks;
    }
    max_weeks[0]=temp;

    temp=avail[countweeks[0]].weeks;
    for(p=countweeks[0];p<countweeks[0]+countweeks[1]-1;p++)
    {
        if(avail[p+1].weeks>=temp) temp=avail[p+1].weeks;
    }
    max_weeks[1]=temp;

    temp=avail[countweeks[0]+countweeks[1]].weeks;
```

```
for(p=countweeks[0]+countweeks[1];p<countweeks[0]+countweeks[1]+co
untweeks[2]-1;p++)
    {
        if(avail[p+1].weeks>=temp) temp=avail[p+1].weeks;
    }
    max_weeks[2]=temp;
}
```

```
// Function to provide week availability constraint//


void builddatabase(void)

{
    int phase,week,person;
    char personlmh[3]={'l','m','h'};

    database=(int ****) malloc(sizeof(int ***)*max_phase);
    for(phase=0;phase<max_phase;phase++)
{
        *(database+phase)=(int ***) malloc(sizeof(int
**)*max_weeks[phase]);

 for(week=0;week<max_weeks[phase];week++)
{


*(*(database+phase)+week)=(int **) malloc(sizeof(int *)*3);

for(person=0;person<3;person++)
{
                    *(*(*(database+phase)+week)+person)=(int *)
malloc(sizeof(int)*count[phase][person]);
                }
            }
    }

for(i=0;i<3;i++)
    {
        loop[i]=count[i][0]+count[i][1]+count[i][2];
    }

     //Initialize the data structure//
for(phase=0;phase<max_phase;phase++)
```

```
for(week=0;week<max_weeks[phase];week++)

for(person=0;person<3;person++)
{
                        if(count[phase][person]==0)
database[phase][week][person][0]=0;
                        else for(i=0;i<count[phase][person];i++)
                                    database[phase][week][person][i]=0;
                  }

    //Fill values//
    for(phase=0;phase<max_phase;phase++)
         for(week=0;week<max_weeks[phase];week++)
              for(person=0;person<3;person++){

                        int z=0,x,offset=0;
                        if(phase==0) offset=0;
                        else for(x=0;x<phase;x++) offset+=loop[x];

                        for(x=0;x<loop[phase];x++)
{

     if(avail[x+offset].exp[0]==personlmh[person]){
                                    if( (week+1) <
avail[x+offset].weeks) database[phase][week][person][z++]=7;
                                    else if( (week+1) ==
avail[x+offset].weeks)
database[phase][week][person][z++]=avail[x+offset].lwd;
                                    else
database[phase][week][person][z++]=0;

                              }
                        }
                  }


}

// Function to evaluate results//

void getuserdata(void)


{
    int flag=1,i,j,k,week,l,z,q;
    int crp[9][3],jump1,jump2;
    float phase_cost, phase_def_det, phase_def_esc, def_exp;
```

```
    float week_cost, week_def_det, week_def_esc;
    float week_chk_hour, phase_chk_hour;
    float week_rew_hour, phase_rew_hour;
    float week_effi, phase_effi;
    float net_cost;
    float def_det[3],def_esc[3];

// Hard coded values of checking rate specifying their optimum
range and prescribed maximum value//

    int opm[3][3]={5, 7, 9,3, 4, 7, 110, 150, 200};


// Hard coded values of effectiveness of low, moderate, high level
experience for normal work product and with normal checking rate//


    float effect_norm[9]={0.35, 0.5, 0.6, 0.32, 0.55, 0.7, 0.3,
0.55, 0.62};

// Hard coded values of effectiveness of low, moderate, high level
experience for complex work product and with above than normal
checking rate//

float effect_comp[9]={0.25, 0.35, 0.5, 0.32, 0.45, 0.6, 0.2, 0.40,
0.55};

float effect_temp[9];

// Hard coded cost of inspectors of low, high, moderate level
experience//


    int cost[3]={100,200,300};
     int done_n,done_c;

// Hard coded value of chances of defect detection in next
phases//


    float p[3]={0.02,0.04,0.01};

// Hard coded values of Defect amplification factors//

float m[3]={2.4, 2.1, 5.1};
    float net_def_det,net_def_esc;
```

```
    FILE *fphr,*fwer;
    fphr=fopen(phase_result,"w");
    fwer=fopen(week_result,"w");
    while(flag)
    {
        for(i=0;i<max_phase;i++) //Phase loop//
        {
label1:  phase_cost=0;
               phase_def_det=0;
               phase_def_esc=0;
               phase_chk_hour=0;
               phase_rew_hour=0;
               phase_effi=0;
               done_n=0;
        done_c=0;
               printf( "\n==For %s Phase== \n",rdc[i]);
               for(week=0;week<max_weeks[i];week++) //Week loop//
               {
                    week_cost=0;
                    week_def_det=0;
                    week_def_esc=0;
                    week_chk_hour=0;
                    week_rew_hour=0;

                    week_effi=0;
                    printf("\n\n==For Week %d ==\n\n",week+1);

                    for(z=0;z<3;z++)
                                        {
                              if(count[i][z]>1){

printf("%s person(s)= %d for    ",lmh[z],count[i][z]);
                                   for(q=0;q<count[i][z];q++)

printf("[%d]  ",database[i][week][z][q]);

printf("days\n");
                              }
                              else

 printf("%s person(s)= %d for %d
days\n",lmh[z],count[i][z],database[i][week][z][0]);
                    }

                         for(j=0;j<3;j++)  //person loop
                    {
                         printf("%s persons :",lmh[j]);
```

```
                        scanf("%d",&persons[3*i+j]);
                        if(persons[3*i+j]!=0)
                        {
                             for(k=0;k<persons[3*i+j];k++)
                             {

                                       jump1=0;jump2=0;

                                       if((so_n[i]-done_n)<=0){
                             printf("Normal pages/loc done!\n");
                                            crp[k][1]=0;
                                            jump1=1;
                                       }

                                       if((so_c[i]-done_c)<=0){
                             printf("Complex pages/loc done!\n");
                                            crp[k][2]=0;
                                            jump2=1;
                                       }

//Condition to check the status of Job//

                                       if((jump1==1)&&(jump2==1)) {
                        printf("for this phase - Job over!\n");
                                            i++;
                                            goto label1;
                                       }

printf("Normal pages/loc left :%d\n",so_n[i]-done_n);
printf("Complex pages/loc left :%d\n",so_c[i]-done_c);

printf("For person %d, enter checking rate [Optimum: %d-%d, Max:
%d]:",k+1,opm[i][0],opm[i][1],opm[i][2]);

scanf("%d",&crp[k][0]);

    // Condition to check specified checking rate//

                                       if(crp[k][0]>opm[i][1])

copyarray(effect_temp,effect_comp);

                                       else
copyarray(effect_temp,effect_norm);

                                       if(jump1!=1){
```

```
                                    printf("Enter normal pages/loc: ");
                                          scanf("%d",&crp[k][1]);
                                          }

                                          if(jump2!=1)
                                                          {
                               printf("Enter complex pages/loc: ");

scanf("%d",&crp[k][2]);
                                          }

lmh_p[j]=crp[k][1];

done_n+=crp[k][1];
done_c+=crp[k][2];

week_chk_hour+=((crp[k][1]/crp[k][0])+(crp[k][2]/crp[k][0]));

week_cost+=((crp[k][1]/crp[k][0])*cost[j]+(crp[k][2]/crp[k][0])*co
st[j]);

phase_chk_hour+=((crp[k][1]/crp[k][0])+(crp[k][2]/crp[k][0]));

phase_cost+=((crp[k][1]/crp[k][0])*cost[j]+(crp[k][2]/crp[k][0])*c
ost[j]);

net_cost+=((crp[k][1]/crp[k][0])*cost[j]+(crp[k][2]/crp[k][0])*cos
t[j]);

                          if(i==0)
def_exp=def_inj_rate[i]*(crp[k][1]+crp[k][2]);//req
                          if (i==1)
def_exp=def_inj_rate[i]*(crp[k][1]+crp[k][2])+(m[i]*phase_def_esc)
;//des
                          if (i==2)
def_exp=def_inj_rate[i]*(crp[k][1]+crp[k][2])+(m[i]*(1-
p[1])*phase_def_esc)+(m[i]*phase_def_esc);//code
week_def_det+=effect_temp[3*i+j]*def_exp;
week_rew_hour+=0.3*effect_temp[3*i+j]*def_exp;
phase_def_det+=effect_temp[3*i+j]*def_exp;
phase_rew_hour+=0.3*effect_temp[3*i+j]*def_exp;

if(i==0)

def_det[i]+=effect_temp[3*i+j]*def_exp;

if(i==1)
```

```
def_det[i]+=effect_temp[3*i+j]*def_exp+(m[0]*phase_def_esc*p[0]);
if(i==2)

def_det[i]+=effect_temp[3*i+j]*def_exp+(((1-

p[0])*m[0]*phase_def_esc)*m[2]*p[2]) +(m[1]*phase_def_esc*p[1]);


week_def_esc+=(1-effect_temp[3*i+j])*def_exp;
phase_def_esc+=(1-effect_temp[3*i+j])*def_exp;

if(i==0)

def_esc[i]+=(1-effect_temp[3*i+j])*def_exp;

if(i==1)

def_esc[i]+=(1-effect_temp[3*i+j])*def_exp+((1-
p[0])*m[0]*phase_def_esc);

if(i==2)

def_esc[i]+=(1-effect_temp[3*i+j])*def_exp+(1-p[2])*m[2]*(1-
p[0])*m[0]*phase_def_esc)+((1-p[1])*m[1]*phase_def_esc);
                                    }
                        }
                }
                //Superuser intervention//
                printf("For evaluation by super user :\n");

for(l=0;l<3;l++)
                {
                        if(persons[3*i+l]==0) continue;
printf("Enter fraction of pages/loc to be checked [%age] for
%s:",lmh[l]);
                        scanf("%f",&su_f[l]);
                        printf("Enter checking rate :");
                        scanf("%d",&su_cr[l]);
                        su_p[l]=su_f[l]*lmh_p[l];
                        su_c[l]=(su_p[l]/su_cr[l])*su_cph/100;
                        net_su_c+=su_c[l];

 printf("Super user cost for %s person : %f\n",lmh[l],su_c[l]);
                }
```

```
 printf("Net super user cost for %d week: %f\n:", week+1,
net_su_c);
                    //Write week results//


fprintf(fwer,"For %s Phase   : \n",rdc[i]);

fprintf(fwer,"For week %d\n",week+1);

fprintf(fwer,"Cost:                  : %f\n",week_cost);

fprintf(fwer,"Check hour   : %f\n",week_chk_hour);

fprintf(fwer,"Rework hour   : %f\n",week_rew_hour);

fprintf(fwer,"Defects detected : %f\n",week_def_det);

fprintf(fwer,"Defects escaped   : %f\n",week_def_esc);

fprintf(fwer,"Efficiency   :   %f\n",week_def_det/week_chk_hour);
                    fprintf(fwer,"\n");

               } //Week ends//

               nc=net_cost;
               A=def_det[0];B=def_esc[0];
               C=def_det[1];D=def_esc[1];
               E=def_det[2];F=def_esc[2];

// Write Phase results//


fprintf(fphr,"For %s Phase   : \n",rdc[i]);

fprintf(fphr,"Cost:                  : %f\n",phase_cost);

fprintf(fphr,"Check hour   : %f\n",phase_chk_hour);

fprintf(fphr,"Rework hour   : %f\n",phase_rew_hour);

fprintf(fphr,"Defects detected : %f\n",phase_def_det);

fprintf(fphr,"Defects escaped   : %f\n",phase_def_esc);

fprintf(fphr,"Efficiency: %f\n",phase_def_det/phase_chk_hour);
```

```
fprintf(fphr,"\n");

printf("\n== For %s phase ==\n",rdc[i]);

 printf("Cost                : %f\n",phase_cost);

 printf("Defects detected: %f\n",phase_def_det);

  printf("Defects escaped : %f\n",phase_def_esc);

 printf("Efficiency          : %f\n",phase_def_det/phase_chk_hour);
        printf("______________________________\n\n");

          } //Phase ends//
        fclose(fphr);
        fclose(fwer);
        printf("\n\n\nplay more.. [y/n]: ");
        if(getch()=='n') flag=0;
    }

// Show end results//

net_def_det=A + C + E + m[0]*B*p[0] + m[1]*p[1]*D + m[2]*p[2]*((1-
p[0])*m[0]*B);
net_def_esc=F + D*m[1]*(1-p[1]) + m[2]*(1-p[2])*((1-
p[0])*m[0])*B);
printf("Net cost: %f\n",nc);
printf("Total defects detected: %f\n",net_def_det);
printf("Total defectes escaped: %f\n",net_def_esc);
}

void copyarray(float a[9],float b[9])
{
     int i;
for(i=0;i<9;i++)
a[i]=b[i];
}
```

APPENDIX 'C'

Input file availability of Inspectors'

ID	EXP. Level	From Date	To Date
200	m	01 02 00	26 04 00
300	h	01 02 00	14 03 00
400	m	01 02 00	15 02 00
500	h	10 04 00	24 04 00
600	h	10 06 00	12 07 00
800	m	07 06 00	30 06 00
900	l	12 04 00	12 07 00

Input file project Schedule

Phase	Start Date	End Date
Reqment	01 02 00	20 02 00
Design	11 04 00	03 05 00
Code	05 06 00	01 07 00

Number of normal requirement document in pages: 300
Number of complex requirement document in pages: 300
Number of normal design document in pages: 400
Number of normal design document in pages: 400
Number of normal line of code: 10000
Number of complex line of code: 9000

Defect Injection Rate:
Requirement = 1.2 per Page
Design = 1.5 per Page
Coding = 0.0049 per Loc

For Reqment Phase :
Cost: : 16315.476563
Check hour : 78.595238
Rework hour : 99.900009
Defects detected : 333.000031
Defects escaped : 387.000031
Efficiency : 4.236898
For Design Phase :
Cost: : 17681.548828
Check hour : 87.065483
Rework hour : 115.488007
Defects detected : 737.555115
Defects escaped : 4614.490723
Efficiency : 8.471269
For Coding Phase :
Cost: : 17953.939453
Check hour : 107.239395
Rework hour : 9.130170
Defects detected : 930.068909
Defects escaped : 15523.995117
Efficiency : 8.672829
Net Cost 194717
Net defect detected 2000.62
Net defect escaped 15523

For Reqment Phase :
For week 1
Cost: : 8169.047852
Check hour : 41.789684
Rework hour : 50.580006
Defects detected : 168.600006
Defects escaped : 222.600006
Efficiency : 4.034489

For Reqment Phase :
For week 2
Cost: : 5301.785645
Check hour : 27.323412
Rework hour : 36.540001
Defects detected : 121.800003
Defects escaped : 121.800003
Efficiency : 4.457716

For Reqment Phase :
For week 3
Cost: : 2844.642822
Check hour : 9.482142
Rework hour : 12.780002
Defects detected : 42.600002
Defects escaped : 42.600002
Efficiency : 4.492656

For Design Phase :
For week 1
Cost: : 7342.261719
Check hour : 37.529762
Rework hour : 54.067505
Defects detected : 180.225006
Defects escaped : 224.774994
Efficiency : 4.802189

For Design Phase :
For week 2
Cost: : 7114.285645
Check hour : 23.714285
Rework hour : 42.210003
Defects detected : 140.699997
Defects escaped : 69.299995

Efficiency : 5.933133

For Design Phase :
For week 3
Cost: : 3225.000000
Check hour : 25.821428
Rework hour : 19.210501
Defects detected : 64.034996
Defects escaped : 156.465012
Efficiency : 2.479917

For Coding Phase :
For week 1
Cost: : 4077.272705
Check hour : 21.772728
Rework hour : 1.987440
Defects detected : 6.624800
Defects escaped : 10.035200
Efficiency : 0.304271

For Coding Phase :
For week 2
Cost: : 7194.444336
Check hour : 47.777779
Rework hour : 3.381000
Defects detected : 11.270000
Defects escaped : 30.380001
Efficiency : 0.235884

For Coding Phase :
For week 3
Cost: : 3488.888916
Check hour : 16.222221
Rework hour : 1.669920
Defects detected : 5.566400
Defects escaped : 7.859600
Efficiency : 0.343134

For Coding Phase :
For week 4
Cost: : 3193.333252
Check hour : 21.466667
Rework hour : 2.091810
Defects detected : 6.972700
Defects escaped : 9.491300
Efficiency : 0.324815

Next strategy:

For Reqment Phase :
Cost: : 16315.476563
Check hour : 78.595238
Rework hour : 99.900009
Defects detected : 333.000031
Defects escaped : 387.000031
Efficiency : 4.236898
For Design Phase :
Cost: : 17681.548828
Check hour : 87.065483
Rework hour : 115.488007
Defects detected : 737.555115
Defects escaped : 4614.490723
Efficiency : 8.471269
For Coding Phase :
Cost: : 17953.939453
Check hour : 107.239395
Rework hour : 9.130170
Defects detected : 930.068909
Defects escaped : 15523.995117
Efficiency : 8.672829
Net cost: 460547
Net defect detected: 5062.14
Net defect Escaped:42202

Weekly Results

For Reqment Phase :
For week 1
Cost: : 8169.047852
Check hour : 41.789684
Rework hour : 50.580006
Defects detected : 168.600006
Defects escaped : 222.600006
Efficiency : 4.034489
For Reqment Phase :
For week 2
Cost: : 5301.785645
Check hour : 27.323412
Rework hour : 36.540001
Defects detected : 121.800003
Defects escaped : 121.800003
Efficiency : 4.457716
For Reqment Phase :
For week 3
Cost: : 2844.642822
Check hour : 9.482142
Rework hour : 12.780002
Defects detected : 42.600002
Defects escaped : 42.600002
Efficiency : 4.492656
For Design Phase :
For week 1
Cost: : 7342.261719
Check hour : 37.529762
Rework hour : 54.067505
Defects detected : 180.225006
Defects escaped : 224.774994
Efficiency : 4.802189
For Design Phase :
For week 2
Cost: : 7114.285645
Check hour : 23.714285
Rework hour : 42.210003
Defects detected : 140.699997
Defects escaped : 69.299995
Efficiency : 5.933133
For Design Phase :

For week 3
Cost: : 3225.000000
Check hour : 25.821428
Rework hour : 19.210501
Defects detected : 64.034996
Defects escaped : 156.465012
Efficiency : 2.479917
For Coding Phase :
For week 1
Cost: : 4077.272705
Check hour : 21.772728
Rework hour : 1.987440
Defects detected : 6.624800
Defects escaped : 10.035200
Efficiency : 0.304271
For Coding Phase :
For week 2
Cost: : 7194.444336
Check hour : 47.777779
Rework hour : 3.381000
Defects detected : 11.270000
Defects escaped : 30.380001
Efficiency : 0.235884
For Coding Phase :
For week 3
Cost: : 3488.888916
Check hour : 16.222221
Rework hour : 1.669920
Defects detected : 5.566400
Defects escaped : 7.859600
Efficiency : 0.343134
For Coding Phase :
For week 4
Cost: : 3193.333252
Check hour : 21.466667
Rework hour : 2.091810
Defects detected : 6.972700
Defects escaped : 9.491300
Efficiency : 0.324815

Next Strategy:

For Reqment Phase :
Cost: : 18793.808594
Check hour : 88.076981
Rework hour : 95.850006
Defects detected : 319.500000
Defects escaped : 400.500000
Efficiency : 3.627509
For Design Phase :
Cost: : 29673.808594
Check hour : 153.130951
Rework hour : 172.016998
Defects detected : 1106.485596
Defects escaped : 7122.436035
Efficiency : 7.225748
For Coding Phase :
Cost: : 24304.496094
Check hour : 122.491547
Rework hour : 11.219040
Defects detected : 1115.178955
Defects escaped : 20213.144531
Efficiency : 9.104130

Weekly Results

For Reqment Phase :
For week 1
Cost: : 9104.166992
Check hour : 44.597221
Rework hour : 38.250000
Defects detected : 127.500000
Defects escaped : 208.500000
Efficiency : 2.858923
For Reqment Phase :
For week 2
Cost: : 5622.976074
Check hour : 29.924208
Rework hour : 38.160004
Defects detected : 127.200005
Defects escaped : 127.200005
Efficiency : 4.250739
For Reqment Phase :
For week 3
Cost: : 4066.666748
Check hour : 13.555555
Rework hour : 19.440002
Defects detected : 64.800003
Defects escaped : 64.800003
Efficiency : 4.780328
For Design Phase :
For week 1
Cost: : 9309.523438
Check hour : 52.809525
Rework hour : 57.955502
Defects detected : 193.184998
Defects escaped : 252.315002
Efficiency : 3.658147
For Design Phase :
For week 2
Cost: : 12264.285156
Check hour : 49.035713
Rework hour : 68.535004
Defects detected : 228.449997
Defects escaped : 176.550003
Efficiency : 4.658849

For Design Phase :
For week 3
Cost: : 8100.000000
Check hour : 51.285713
Rework hour : 45.526501
Defects detected : 151.754990
Defects escaped : 301.244995
Efficiency : 2.959011
For Coding Phase :
For week 1
Cost: : 7763.713867
Check hour : 38.611500
Rework hour : 3.874920
Defects detected : 12.916400
Defects escaped : 16.483601
Efficiency : 0.334522
For Coding Phase :
For week 2
Cost: : 5938.888672
Check hour : 27.388889
Rework hour : 2.634240
Defects detected : 8.780800
Defects escaped : 14.739200
Efficiency : 0.320597
For Coding Phase :
For week 3
Cost: : 10601.894531
Check hour : 56.491161
Rework hour : 4.709880
Defects detected : 15.699599
Defects escaped : 24.480400
Efficiency : 0.277912

Next strategy	For Reqment Phase :
	Cost: : 12466.450195
	Check hour : 64.692276
	Rework hour : 78.696014
	Defects detected : 262.320007
	Defects escaped : 503.280029
	Efficiency : 4.054889
	For Design Phase :
	Cost: : 23573.570313
	Check hour : 147.902374
	Rework hour : 138.415512
	Defects detected : 1027.348511
	Defects escaped : 7445.123535
	Efficiency : 6.946126
	For Coding Phase :
	Cost: : 22978.894531
	Check hour : 118.411743
	Rework hour : 11.560081
	Defects detected : 1219.876343
	Defects escaped : 21497.970703
	Efficiency : 10.301988
	Net cost: 100110
	Net defect detected: 2509
	Net defect escaped: 21497

Weekly Results:

For Reqment Phase :
For week 1
Cost: : 6511.111328
Check hour : 34.000000
Rework hour : 35.640003
Defects detected : 118.800003
Defects escaped : 200.400009
Efficiency : 3.494118
For Reqment Phase :
For week 2
Cost: : 3376.767578
Check hour : 18.156565
Rework hour : 29.106003
Defects detected : 97.020004
Defects escaped : 228.180008
Efficiency : 5.343522
For Reqment Phase :
For week 3
Cost: : 2578.571289
Check hour : 12.535714
Rework hour : 13.950002
Defects detected : 46.500000
Defects escaped : 74.700005
Efficiency : 3.709402
For Design Phase :
For week 1
Cost: : 5842.856934
Check hour : 39.595238
Rework hour : 37.966499
Defects detected : 126.555000
Defects escaped : 258.945007
Efficiency : 3.196218
For Design Phase :
For week 2
Cost: : 9696.428711
Check hour : 54.821430
Rework hour : 57.262501
Defects detected : 190.875000
Defects escaped : 184.125000

Efficiency : 3.481759
For Design Phase :
For week 3
Cost: : 8034.285645
Check hour : 53.485714
Rework hour : 43.186501
Defects detected : 143.955002
Defects escaped : 295.545013
Efficiency : 2.691466
For Coding Phase :
For week 1
Cost: : 7515.895996
Check hour : 39.813927
Rework hour : 3.683820
Defects detected : 12.279400
Defects escaped : 18.590601
Efficiency : 0.308420
For Coding Phase :
For week 2
Cost: : 5791.666504
Check hour : 30.138889
Rework hour : 3.089940
Defects detected : 10.299800
Defects escaped : 14.690200
Efficiency : 0.341745
For Coding Phase :
For week 3
Cost: : 5375.000000
Check hour : 17.916666
Rework hour : 2.551920
Defects detected : 8.506400
Defects escaped : 5.213600
Efficiency : 0.474776
For Coding Phase :
For week 4
Cost: : 4296.331543
Check hour : 30.542265
Rework hour : 2.234400
Defects detected : 7.448000
Defects escaped : 15.582000
Efficiency : 0.243859

Date Slip

The book is to be returned on the date last stamped.